THE UNIVERSITY *of* EDINBURGH

Scheme-based Theorem Discovery and Concept Invention

Omar Montaño Rivas

Doctor of Philosophy

Centre for Intelligent Systems and their Applications
School of Informatics
University of Edinburgh

2012

Abstract

In this thesis we describe an approach to automatically invent/explore new mathematical theories, with the goal of producing results comparable to those produced by humans, as represented, for example, in the libraries of the Isabelle proof assistant. Our approach is based on 'schemes', which are formulae in higher-order logic. We show that it is possible to automate the instantiation process of schemes to generate conjectures and definitions. We also show how the new definitions and the lemmata discovered during the exploration of a theory can be used, not only to help with the proof obligations during the exploration, but also to reduce redundancies inherent in most theory-formation systems. We exploit associative-commutative (AC) operators using ordered rewriting to avoid AC variations of the same instantiation. We implemented our ideas in an automated tool, called IsaScheme, which employs Knuth-Bendix completion and recent automatic inductive proof tools. We have evaluated our system in a theory of natural numbers and a theory of lists.

Acknowledgements

First and foremost I offer my sincerest gratitude to Roy McCasland, Alan Bundy and Lucas Dixon for their exceptional supervision of this thesis. Their intuitions and directions have been consistently a source of inspiration to me.

A very special thanks to Alan Smaill and Jacques Fleuriot for their feedback of my work. Without them, this thesis as it is would not exist. Similarly, many thanks to Moa Johansson, Gudmund Grov and Ewen Maclean for their comments and useful remarks. I am also very grateful to Iain Whiteside for proof-reading parts of this thesis.

I have very much enjoyed working together with Iain, Michael Chan, Petros Papapanagiotou, Phil Scott, Liwei Deng, Alison Pease, Fiona McNeill and all the members of the DReaM group. Their friendship and professional collaboration meant a great deal.

Thanks to Bruno Buchberger, Adrian Craciun and Temur Kutsia for pointing me to various relevant papers and their feedback of my work during my visits in Hagenberg. Also, thanks to the rest of the people at RISC I met during my visits because they always made me feel welcome.

I wish to thank my mother and my brother for their confidence, but most importantly for always being there. Finally, I would like to thank my wife Laura for her unconditional love, support and for giving me a beautiful son.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Omar Montaño Rivas*)

Publications

Parts of chapters 2, 3, 4 and 7 have appeared in the Journal of Expert Systems with Applications (ESWA) [54].

Parts of chapters 4 and 7 have appeared in the Proceedings of the 9th Mexican international conference on Advances in artificial intelligence and won the Best Paper Award [53].

Table of Contents

Chapter 1

Introduction

Human mathematical discovery processes include the invention of definitions, conjectures, theorems, examples, counter-examples, problems and algorithms for solving these problems. Automating these discovery processes is an exciting area for research with applications to automated theorem proving [32, 64], algorithm synthesis [5, 7], and others [42, 15, 48].

A variety of theory-exploration computer programs have been implemented [40, 15, 48] and different approaches have been identified [64]. A recent approach, scheme-based mathematical theory-exploration [6], has been proposed and its implementation is being undertaken within the *Theorema* project [8]. The central idea of the approach is that of a scheme; i.e. a higher-order formula intended to capture the accumulated experience of mathematicians for discovering new pieces of mathematics (see section 2.2).

A case study of mathematical theory-exploration is described in [20] for the theory of natural numbers using the scheme-based approach. However, apart from this paper there is, to our knowledge, no other case study of scheme-based mathematical theory-exploration. In the Theorema system, which was used to carry out the aforementioned case study, the user had to provide the appropriate substitutions (Theorema cannot perform the possible instantiations automatically). The authors also pointed out that the implementation of some provers was still in progress and that the proof obligations were in part 'pen-and-paper'. From these observations, a natural questions arises: can we effectively mechanise the process of selecting the instantiations for schemes, and subsequently automate the proof obligations induced by the conjectures and definitions?

This thesis gives a positive answer to the above question. We show that it is possi-

ble to automate the instantiation process of schemes to generate conjectures and definitions. This automation introduces an unavoidable combinatorial explosion during the exploration of a theory and a central goal of this thesis aims to tackle this problem (see chapter 4). We also show how the new definitions and the lemmata discovered during the exploration of a theory can be used, not only to help with the proof obligations during the exploration, but also to reduce redundancies inherent in most theory-formation systems.

We begin this thesis by first discussing the initial motivations for the project in 1.1. Then we provide a description of the aims of the project (section 1.2), followed by the contributions this work makes to state of the art (section 1.3) and finally the organisation of this thesis (section 1.4).

1.1 Motivation

The logical and inductive consequences of a set of consistent axioms form the theory of those axioms. In such a theory there are many boring theorems and scattered among them there are a few interesting ones. The few interesting ones include those considered as theorems by human experts in the domain, which get printed in journals, books, or presented at conferences. Although humans have identified many interesting theorems (of a given set of axioms), it seems inevitable that there are more out there. The problem is thus to generate and identify these undiscovered interesting theorems.

Automating the process of theorem selection is, however, a difficult task. In the computer science and artificial intelligence communities, there have been different heuristics used in different systems. For example, MATHsAiD (see section 2.1.3) uses *non-triviality* to get only theorems not 'trivially' produced by prior theorems, *irredundancy* is used to determine whether a set of hypotheses are required for a given conclusion and *simplicity* ensures that the theorem is simple. In [18] the authors surveyed five mathematical discovery programs to highlight how they estimate the 'interestingness' of concepts and conjectures and extracted some common notions. *Empirical plausibility of conjectures* is defined as a heuristic to avoid conjectures which have known counterexamples; *novelty* avoids repetition (very common in search); *surprisingness* avoids tautologies; *applicability* uses the number of examples the concept has to determine how important it is (more examples means more important); *comprehensibility and complexity*: more comprehensible means more interesting; finally, *utility* is the technique to enable the user (or an external automated reasoning program) to express

a particular interest to the mathematical discovery program.

Automated theory-exploration consists not only of inventing mathematical theorems from a set of axioms. Among other activities, it also includes the discovery of new concrete notions or definitions. This activity helps with the development of the theory in a coherent and natural way in which exploration proceeds in layers. For example, starting from the concepts of zero and succesor we can define addition. Once this new concept of interest is added to the theory, we can start guessing its properties by conjecturing statements about it. Afterwards, we begin a new layer in which multiplication is defined in terms of addition, followed by another layer with exponentiation using the definition of multiplication and so on. This process is similar to the well-known programming paradigm in computer science that programs should be developed incrementally. In this paradigm, a programmer defines a collection of data structures and basic functions and then proceeds by defining new functions in terms of the given ones. Eventually, the extended system acts itself as a base system which is further extended by new functions and so on.

A recent approach to automate these processes is *scheme-based mathematical theory-exploration* [6]. However, this new approach has not been extensively explored, particularly in regards to the problem of theorem discovery and concept invention. In [20], Madalina Hodorog and Adrian Craciun showed interesting results – albeit produced partly by hand – with the theory of natural numbers that may well be applied to the problem of mathematical theory-exploration in other theories.

Having described a general scenario for mathematical theory-exploration, we can provide some motivations for the project. Firstly, to explore the capabilities and limitations of the scheme-based approach applied to the problem of theorem discovery and concept invention. Secondly, to give a rational reconstruction on the techniques and heuristics used to automate these processes.

Thirdly, mathematical theory-exploration is an important and difficult problem with applications in other areas. In automated theorem proving, theory-exploration can be used to explore a mathematical theory before any proof attempt, providing lemmata which could be exploited to improve proof automation [32, 64]. In algorithm synthesis, theory-exploration could be used to generate algorithms with explicit specifications [5, 7].

1.2 Aims of the project

This research was focused on the process of inventing mathematical concepts and inventing (and verifying) conjectures about those concepts. The hypotheses of the project were:

- A theory-exploration framework based on schemes can be used for the problem of theorem discovery and concept invention.
- There exists a small number of schemes that consistently instantiate to a large number of theorems and/or concepts that formal methods practitioners generally find to be interesting, while maintaining a relatively small quantity of uninteresting theorems and concepts.

In order to evaluate our hypotheses we aimed to design and develop an automated computer system that uses the scheme-based approach as its primary method for synthesising theorems and concepts. The system constructed must satisfy the following criteria:

1. The system works in a range of mathematical domains. The representational technicalities of axioms and schemes should not depend on a particular domain or theory.
2. The inputs to the system are a mathematical theory, a set of schemes, and a proof technique for the proof obligations.
3. Interaction with the system is allowed but it must be possible to distinguish the work done by the system in contrast to that done by the user.
4. The proof of theorems can be done automatically or interactively.

To demonstrate that the system met the above criteria, we needed to employ the system in a number of mathematical theories; this would demonstrate the applicability (generality) of the framework developed. We decided to perform a precision / recall analysis with Isabelle's libraries [59], in order to evaluate the quality of the theorems and definitions found by our method and implementation. The theories we chose to use were natural numbers, lists and trees because they are known to be challenging not only for theory-formation, but also for inductive theorem proving [10, 23, 32].

1.3 Contributions

- We designed and implemented the IsaScheme program following a new approach to mathematical theory-exploration based on schemes. Other than IsaScheme, Theorema is the only system performing the exploration of mathematical theories based on schemes [8]. However, in the latter system, the user needs to perform all schematic substitutions manually, as Theorema does not instantiate the schemes automatically from a set of terms. In IsaScheme this process was completely automated. Another important difference is that ensuring the soundness of definitions is left to the user in Theorema. IsaScheme was developed on top of Isabelle/HOL following the LCF tradition. By using existing definitional tools we ensured that IsaScheme is conservative by construction, and thus offered a maximum of safety from unsoundness [39].

- IsaScheme has been successfully applied to different mathematical theories, such as: natural numbers, lists and binary trees. We performed a successful precision/recall analysis with Isabelle's theory library as reference to evaluate the quality of the theorems and definitions found.

- We designed and implemented a novel technique based on termination and completion of rewrite systems to show how the new definitions and the lemmata discovered during the exploration of a theory can be used, not only to help with the proof obligations during the exploration, but also to reduce redundancies inherent in most theory-formation systems. This technique allowed us to neglect most of the existing heuristics used in mathematical theory-exploration, such as those mentioned in [18] (see sections 2.1.2 and 7.6.4). This technique also helped to invent useful lemmas (critical pairs) during completion (see sections 3.4, 4.3 and 4.8).

1.4 Layout of Thesis

The organization of the thesis is as follow:

Chapter 2 We report a summary of work on the topics related to this thesis. In particular, automated theorem discovery and theory-formation systems, scheme-based mathematical theory-exploration, and the Isabelle interactive theorem prover.

Chapter 3 We present background material on topics which are related to our work, including higher-order rewrite systems (HRS), higher-order recursive path orders, confluence and completion of rewrite systems.

Chapter 4 We describe how we can soundly automate the instantiation process of schemes on top of the simply-typed lambda calculus of Isabelle/HOL [55]. We show how new definitions and the lemmata discovered during the exploration of the theory can be used not only to strengthen Isabelle's tools, such as the *Simplifier* [56] or the *function package* [39], but also to reduce redundancies inherent in most theory-formation systems. We also describe our theory-exploration algorithms where the processes of theorem and definition discovery are linked together.

Chapter 5 We describe a simple inductive proof technique for Isabelle that has been shown to be very useful when combined with the term rewrite system described in Chapter 4.

Chapter 6 We conduct several case studies in the theory of natural numbers, the theory of lists and a theory of trees to evaluate how similar are the results obtained with our method and implementation to those in the libraries of the Isabelle proof assistant. We also relate several aspects of our research to the work of other researchers. One of these aspects is a comparison between IsaScheme and other theory-exploration tools.

Chapter 7 We summarise the work in this thesis, and describe possible directions for future research.

There are four appendices: Appendix A provides the datatypes and definitions used in the thesis. Appendix B presents two sessions using IsaScheme with the naturals. Appendix C contains some experimental results in inductive theorem proving. Appendix D contains some of the theorems discovered by IsaScheme.

Chapter 2

Literature Survey

2.1 Theory-Exploration Systems

Probably the first computer program performing theory-formation was Lenat's AM [41]. AM is one of the most widely cited programs in Artificial Intelligence (despite the fact that there has been controversy about the methodology Lenat used in the construction of AM and whether his reported work was accurate enough for his research to be reproduced by other scientists [61, 15]). Although our research is not based on Lenat's work, we survey his AM system in section 2.1.1 for historical reasons.

An important approach for automated discovery in mathematics is to use examples as a primary source for invention, namely the *inductive approach*[1] [64]. The idea behind this approach is to use particular instances (of some sort) to construct mathematical conjectures, e.g. if we observe that 3, 5, 7, 11 are all odd and prime numbers, then a possible conjecture could be to say that all odd numbers are primes. An attempt is then made to prove or disprove this conjecture and if disproved, modifications to the conjecture are made. A theory-formation system using an example driven approach is HR [15], which we briefly describe in section 2.1.2.

The inductive approach has the disadvantage of generating false conjectures, as is the case with the conjecture about odd and prime numbers. A theory-formation technique that does not have this drawback is the so-called *deductive approach* [64]. The deductive approach only generates theorems, and the advantage is that each generated theorem does not pass through the conjecture making stage. Every output is known to be a theorem and thus counter-example checking is unnecessary. Section 2.1.3 surveys

[1] Note that here we are referring to philosophical induction or inductive learning, not to be confused with mathematical induction.

the MATHsAiD system, which works using the deductive approach.

One possible form of invention is to create conjectures by mechanical manipulation of symbols. This syntactical technique is often called *generative approach* [64], and is used by the MCS [25] system and by IsaCoSy [32], which are described in sections 2.1.4 and 2.1.5, respectively.

Other than IsaScheme, Theorema is the only system performing the exploration of mathematical theories based on schemes. This approach for theory-exploration was proposed by Bruno Buchberger and has been described in several papers [6, 8, 7]. We briefly describe Buchberger's approach in sections 2.1.6 and 2.2. Finally, we also survey the interactive theorem prover Isabelle in section 2.3.

2.1.1 AM

In [41] Douglas Lenat outlined the AM program. This program was able to discover relationships between known concepts and to define new ones from old ones. Starting with 115 concepts such as relations, lists, sets and bags, the program would discover number theory concepts and conjectures such as prime numbers, Diophantine equations, unique factorisation of numbers into primes and Goldbach's Conjecture. In set theory, AM would discover concepts and conjectures such as De Morgan's Law, singletons, subsets and supersets.

Mathematical concepts were represented as a list of 25 slots or facets. Each facet corresponded to some aspect of a concept, e.g. the definition for the concept, an algorithm (in LISP) for calculating examples of the concept, examples of the concepts, which other concepts it was a generalization of, conjectures involving the concept, etc.

The basic discovery activity was to choose some facets of some concept, and then try to fill in new entries to store there; this would occasionally cause new concepts to be defined. The high level decision about which facet of which concept to work on next could be handled by maintaining an ordered agenda of such tasks. The techniques for actually carrying out a task were contained within a collection of 242 heuristics and each of them had a well-defined domain of applicability.

2.1.2 HR

HR (named after the mathematicians Hardy and Ramanujan) is a theory-exploration system which uses an example driven approach for theory-formation [15]. Broadly speaking HR works using the model generator MACE [52] to generate objects of in-

terest from a set of examples and definitions, then a process of concept formation and conjecture making is carried out, and finally the Otter [49] theorem prover is used to prove the conjectures [15, 16, 17].

The process of concept invention is carried out from old concepts starting with the concepts provided by MACE at the initial stage. These concepts, stored as data-tables of examples rather than definitions, are passed through a set of 10 production rules whose purpose is to manipulate and generate new data-tables[2]. Specialisation and generalisation of data-tables and ways to combine and negate them are the strategies followed by the production rules. For example, the *Match* rule extracts those rows of the table where the entries in certain columns were equal, the *Exists* rule removes at least one of the columns of the table, the *Compose* rule overlaps the rows of two tables to produce a new one and so on.

Concept-formation is driven as a heuristic search: less interesting concepts are developed after more interesting ones. There are heuristics to measure different aspects of each concept. *Comprehensibility* (or *Complexity*) is used to measure how many production rule steps have been applied to create the concept (simpler concepts were better), *Novelty* measures how many times the categorisation of a concept has been seen (yielding lower scores for categorisations that have been seen many times before), *Parsimony* measures how small or large is the data-table containing the concept (smaller tables are scored better), and finally *Applicability* is the number of examples the concept has calculated as the proportion of entities which appear in the left hand column of the data-table for the concept with respect to the number of entities in the theory.

Theory-formation is built on top of concept formation. HR takes the concepts obtained by the production rules and forms conjectures about them. There are different types of conjectures HR can make, e.g. *equivalence* conjectures which amounts to finding two concepts and stating that their definitions are equivalent, *implication* conjectures are statements relating two concepts by stating that the first is a specialization of the second (all examples of the first will be examples of the second), *non-existence* conjectures are statements that a particular definition is inconsistent with the axioms of the theory (there is no example which satisfies the definition), etc. In addition to finite algebra, number theory and graph theory, HR has also been used for music, visual arts and games.

[2]The number of production rules increased recently to 16 (see [19]).

2.1.3 MATHsAiD

Roy McCasland's MATHsAiD program was intended for use by research mathematicians and was designed to produce interesting theorems from the mathematician's point of view [47]. MATHsAiD starts with an axiomatic description of a theory; hypotheses and terms of interest are then generated (hypothesis generator), logical and inductive consequences of the hypotheses are inferred (theorem generator) and then a filtering process is carried out according to interestingness measures (theorem filter). *Non-triviality* ensures that the theorems are not *trivially* producible by prior theorems. *Irredundancy* is used to determine whether all the given hypotheses are required for a given conclusion. And *simplicity* ensures that the theorem is the simplest in its equivalence class.

Given the set of axioms and definitions, the hypothesis generator (HG) builds up a finite sequence $\{H_i\}_{i=1...n}$ where H_i is a set of hypotheses and a selection of axioms and terms of interest corresponding to each H_i. Each H_i facilitates the discovery of common properties like commutativity or associativity for operators and reflexivity, symmetry and transitivity for relations. Examples of these sequences are found at [45]. The idea behind this was to concentrate on local aspects of the theory and to build theorems in layers, rather than build them all at once. Moreover, the HG also looks at the axioms or theorems for finding reverse implications. If such a reverse implication exists, it is passed over to the theorem generator that will attempt to prove it.

The set of hypotheses H_k constructed by the HG becomes the input of the theorem generator (TG). TG asserts the hypotheses and applies forward deduction using the built-in, first-order theorem prover (MATHsAiD uses Prolog's first-order predicate calculus) to derive all conclusions. Conclusions satisfying non-triviality are then asserted, the process starts over, until no more new conclusions can be found. The resulting conclusions are then passed to the theorem filter.

The theorem filter takes the conclusions from the TG and filters them with the irredundancy and simplicity heuristics. All theorems failing the test are eliminated and all remaining theorems are stored within MATHsAiD.

This approach provided some encouraging results in set theory, number theory and group theory [45]. Theorems that typically appear in books were identified as interesting by MATHsAiD. Furthermore, the system was recently extended towards automated discovery of inductive theorems [46].

2.1.4 MCS

The MCS (Named after the acronym of Model Conjecture Searching) program [25] uses a "generate and test" approach for finding interesting theorems. MCS started with a given theory consisting of a signature and a set of axioms and proceeded with the generation of possible models for the theory using model generators like FINDER [63], SEM [68] or MACE [50]. Then well-formed formulae were constructed and tested against the already generated models. Depending on the results, the set of formulae were divided into three different subsets: S_0 contained the formulas which were false in every model, S_1 contained the formulas which were true in every model and S_2 the rest of the formulas. The formulas in S_0 were discarded while the formulas in S_1 were considered conjectures. Inductive learning was used to find relationships between formulae in S_1, e.g. which formula (or set of formulas) implies other formulae . The formulas in S_2 were processed with machine learning techniques to find possible relationships between them e.g. using the free variables of a pair of terms T_1 and T_2, formulate different combinations of quantifiers Θ and generate the conjecture $\Theta(T_1 = T_2)$. As a last and optional stage, the conjectures were proved or disproved using EQP [51].

2.1.5 IsaCoSy

IsaCoSy is a theory-exploration system for Isabelle/HOL [32]. It generates conjectures in a bottom-up fashion from the signature of an inductive theory. The synthesis process is accompanied by automatic counter-example checking and a proof attempt (using the inductive prover IsaPlanner [24]) in case no counter-example is found by Quickcheck [14]. All theorems found are then used as constraints for the synthesis process; synthesis generates only irreducible terms w.r.t. a rewrite system defined by the discovered theorems. IsaCoSy has been used to synthesise theorems about natural numbers, lists and binary trees [32].

2.1.6 Theorema

Theorema uses a model for theory-exploration based on higher-order formulae or schemata representing prior mathematical knowledge of different types. Given a mathematical theory and a set of schemes, the model would proceed in a bottom up/top down fashion to perform:

- Invention of concepts (i.e. definitions) by using definition schemata.

- Invention of conjectures about the new definitions by using conjecture schemata and proving or disproving them using available proof techniques (so-called *reasoners*).

- Invention of algorithm specifications (so-called *problems*) involving the concepts (i.e. formulae of the form $\forall x.\, Q(x, A(x))$ where $Q(x, y)$ describes the ralation between the input x and the output y, and A is the algorithm to synthesise).

- Invention and verification of algorithms satisfying the specifications.

A complete description of this model is beyond the scope of this thesis as we are only interested in the first two points, i.e. inventing mathematical concepts and inventing (and verifying) conjectures about those concepts (see section 2.2). For the interested reader, an account of the complete model is described in [6, 8, 7].

Parts of this model for theory-exploration have been successfully implemented in Theorema [7]. However, in terms of automation, there are certain key aspects which can be further improved. One such aspect is the automatic instantiation of schemes with the signature of a mathematical theory. Currently, the user has to provide the appropriate substitutions as Theorema cannot perform the possible instantiations automatically [20]. The authors in [20] also reported that the implementation of some provers was still in progress and that the proof obligations were in part 'pen-and-pencil'. There is also room for improvement in Theorema's definitional infrastructure. Ensuring the soundness of definitions is left to the user in Theorema. This can be an important impediment towards automation because an automatic instantiation of schemes could lead to invalid or potentially unsound definitions (see section 2.2).

2.2 Scheme-based Synthesis of Conjectures and Definitions

The central idea of scheme-based mathematical theory-exploration is that of a scheme; i.e. a higher-order formula intended to capture the accumulated experience of mathematicians for discovering new pieces of mathematics. The invention process is carried

out through the instantiation[3] of variables within the scheme. As an example, let $\mathcal{T}_N$ be the theory of natural numbers in which we already have the constant function zero (0), the unary function successor (*suc*) and the binary function addition ($+$) and let *s* be the following scheme which captures the idea of a binary function defined recursively in terms of other functions.

$$\left(\begin{array}{l} def\text{-}scheme\ G\ H\ I\ J \equiv \\ \qquad \exists f.\ \forall xy.\ \ \bigwedge \left\{ \begin{array}{rcl} f\ G\ y & = & H\ y \\ f\ (I\ x)\ y & = & J\ y\ (f\ x\ y) \end{array} \right. \end{array} \right) \tag{2.1}$$

Here the existentially quantified variable f stands for the new function to be defined in terms of the variables G, H, I and J. We can generate the definition of multiplication by allowing the theory $\mathcal{T}_{\mathcal{N}}$ to instantiate the scheme with $\sigma_1 = \{G \mapsto 0, H \mapsto (\lambda x.\ 0), I \mapsto suc, J \mapsto +\}$ (here $f \mapsto *$).

$$\begin{array}{rcl} 0 * y & = & 0 \\ suc\ x * y & = & y + (x * y) \end{array}$$

Multiplication can then in turn be used for the invention of the concept of exponentiation with the substitution $\sigma_2 = \{G \mapsto 0, H \mapsto \lambda x.\ suc\ 0, I \mapsto suc, J \mapsto *\}$ on scheme 2.1 (note that the exponent is the first argument in this case).

$$\begin{array}{rcl} pow\ 0\ y & = & suc\ 0 \\ pow\ (suc\ x)\ y & = & y * pow\ x\ y \end{array}$$

Schemes can be used not only for the invention of new mathematical concepts or definitions, they also can be used for the invention of new conjectures about those concepts. The scheme 2.2 creates conjectures about the *left-distributivity* property of two binary operators in a given theory (the variables P and Q stand for the binary operators). Therefore if we are working w.r.t. $\mathcal{T}_{\mathcal{N}}$ extended with multiplication and exponentiation, we can conjecture the left-distributivity property of multiplication and addition and also between exponentiation and multiplication by using the substitutions $\sigma_3 = \{P \mapsto +, Q \mapsto *\}$ and $\sigma_4 = \{P \mapsto *, Q \mapsto pow\}$ respectively on the scheme (2.2).

$$\left(\begin{array}{l} left\text{-}distributivity\ P\ Q \equiv \\ \qquad \forall xyz.\ Q\ x\ (P\ y\ z) = P\ (Q\ x\ y)\ (Q\ x\ z) \end{array} \right) \tag{2.2}$$

[3] In Theorema, the instantiation process is limited to function, predicate or constant symbols already known in the theory. Additionally, IsaScheme can use any well-formed closed term of the theory including λ-terms such as $(\lambda x.\ x)$.

The aforementioned substitutions give the conjectures

$$\begin{aligned} x*(y+z) &= (x*y)+(x*z) \\ pow\ x\ (y*z) &= (pow\ x\ y)*(pow\ x\ z) \end{aligned}$$

It is important to note that schemes could generate non-terminating definitions and false conjectures. For example, consider the substitution $\sigma_4 = \{G \mapsto 0, H \mapsto (\lambda x.\ 0), I \mapsto (\lambda x.\ x), J \mapsto +\}$[4] on scheme 2.1

$$\begin{aligned} f\ 0\ y &= 0 \\ f\ x\ y &= y + f\ x\ y \end{aligned}$$

This instantiation immediately leads to logical inconsistencies by subtracting $f\ x\ y$ from the second equation producing $0 = y$. This definition is invalid because, contrary to the natural interpretation of I as a constructor symbol, schemes do not express such conditions on instantiations. Similarly, we can also obtain false conjectures from a substitution, e.g. $\sigma_4 = \{P \mapsto *, Q \mapsto +\}$ on scheme 2.2 instantiates to $x+(y*z) = (x+y)*(x+z)$.

2.3 Isabelle prover

A logical framework is a meta-language for the formaliasation of deductive systems so called object logics. Isabelle is a logical framework in form of a generic LCF-style theorem prover [57, 59, 29]. Isabelle's meta-logic, called Isabelle/Pure, only consists of equality $\equiv$, implication $\Longrightarrow$, universal quantification $\bigwedge$, and functional abstraction λ. A number of object logics such as first-order predicate logic, Zermelo-Fraenkel set theory, constructive type-theory, and higher-order logic have been encoded in Isabelle and are part of its standard distribution.

2.3.1 Isabelle/Pure

We describe Isabelle/Pure. Types are built from type variables $(\alpha, \beta, \gamma, \ldots)$ and type constructor symbols written in postfix notation like ML. The function space constructor $\Rightarrow$ is written infix. A special type *prop* denotes propositions, which can be formed using the built-in constants $\Longrightarrow :: prop \Rightarrow prop \Rightarrow prop$, $\bigwedge :: (\alpha \Rightarrow prop) \Rightarrow prop$ and

[4] Note that all theories considered depend upon the simply-typed lambda calculus of Isabelle/Pure. Therefore, $(\lambda x.\ x)$ is a perfectly valid mathematical object. In fact, we can choose to have any (finite) set of well-formed closed terms as the initial theory elements for the exploration of the theory (see section 4.2 for details).

$\equiv :: \alpha \Rightarrow \alpha \Rightarrow prop$. Object logics are defined on top of the meta-logic by declaring their connectives as constants, and their axioms and inference rules as axioms in pure.

Logical rules like

$$\frac{P \quad Q}{P \wedge Q}\ (\text{conjI}) \qquad\qquad \frac{P \wedge Q}{P}\ (\text{conjunct1})$$

are represented in Isabelle by meta-implications

$$?P \Longrightarrow ?Q \Longrightarrow ?P \wedge ?Q \qquad\qquad ?P \wedge ?Q \Longrightarrow ?P$$

The variables prefixed by a question mark are so-called *meta-variables* or *schematic variables*. These variables are allowed to be instantiated by unification or matching.

2.3.2 Isabelle/HOL

Isabelle/HOL is one of the most developed and widely used object-logics in Isabelle. It is based on Church's simple theory of types [13] and it is compatible with the other major implementations of higher-order logic, namely HOL [27] and HOL Light [28].

Object-level propositions are encoded with type *bool*, with the usual connectives $\vee, \wedge, \longrightarrow, \neg$, among others. Object level propositions are coerced to the corresponding meta-level proposition using the symbol $Trueprop :: bool \Rightarrow prop$ (which is usually hidden by the syntax layer). Object level equality is denoted by $= :: \alpha \Rightarrow \alpha \Rightarrow bool$.

2.3.3 Tactical Theorem Proving

Isabelle is a tactical theorem prover in the tradition of LCF [26]. It is implemented in ML which is a strongly typed functional programming language. The type `thm` of Isabelle plays a crucial role in the implementation. Objects of this type can only be generated by a fixed and small number of functions implementing the primitive inference rules of the logical kernel in Isabelle/Pure.

Tactics are ML functions mapping theorems to theorems. Diverse reasoning techniques used in forward or backward proofs are implemented by basic tactics. They are combined with the help of *tacticals* to produce new tactics. Typical examples are choice or repetition of tactics. This yields an elegant and trustworthy way to implement complex proof methods [29].

2.3.4 Simplifier

The simplifier is a powerful "parameterized tactic" in Isabelle [58, 56]. It performs conditional and unconditional higher-order rewriting and uses contextual information (assumptions). It is parametrised by different components where the rewrite rules play an important role. Rewrite rules are theorems expressing some form of equality, for example:

$$\begin{array}{rcl} Suc(?x) + ?y & = & Suc(?x + ?y) \\ (\forall x.\, ?P(x) \wedge ?Q(x)) & \leftrightarrow & (\forall x.\, ?P(x)) \wedge (\forall x.\, ?Q(x)) \\ ?A \cup ?B & \equiv & \{x.\, x \in ?A \vee x \in ?B\} \end{array}$$

Internally, all rewrite rules are translated into meta-equalities. Reflection rules relate the meta-equality ($\equiv$) to object equality ($=$) or equivalences ($\leftrightarrow$). We therefore do not distinguish between these in the presentation.

Conditional rewrites such as $0 < ?n \Longrightarrow abs(?n) = ?n$, are also valid rewrite rules. The conditional rewrites have the form

$$C \Longrightarrow s = t$$

When the simplifier is applied to a subgoal, it replaces subterms of the goal matching s by t, provided it can prove the corresponding instance of C.

2.3.5 Permutative rewrite rules

A rewrite rule is called *permutative* if the left-hand side and right-hand side are the same up to variable renaming [66]. The most common permutative rule is commutativity with the form $P(x,y) = P(y,x)$, e.g.

$$\begin{array}{rcl} x + y & = & y + x \\ x * y & = & y * x \\ max(x,y) & = & max(y,x) \end{array} \tag{2.3}$$

Other examples include $(x - y) - z = (x - z) - y$ and $x\hat{}(y * z) = x\hat{}(z * y)$. These rules are inherently non-terminating and are common enough that the simplifier uses a special strategy on them, called *ordered rewriting*. A permutative rewrite rule is applied only if it decreases the given term with respect to some ordering. By default, the simplifier uses a standard lexicographic ordering on terms (which can be changed

for special applications). For example, (2.3) rewrites $b * a$ to $a * b$ but here the process stops because $a * b$ is strictly less than $b * a$.

In [43] is described how ordered rewriting can be used effectively with AC-operators. For example, the rewrites

$$f(f(x,y),z) = f(x,f(y,z)) \tag{2.4}$$

$$f(x,y) = f(y,x) \qquad \text{if } x > y \tag{2.5}$$

$$f(x,f(y,z)) = f(y,f(x,z)) \qquad \text{if } x > y \tag{2.6}$$

where f is a commutative and associative binary operator, sort a term lexicographically:

$$f(f(b,c),a) \xrightarrow{(2.4)} f(b,f(c,a)) \xrightarrow{(2.5)} f(b,f(a,c)) \xrightarrow{(2.6)} f(a,f(b,c)).$$

2.3.6 Function Package

The function package allows one to safely define general recursive function definitions for Isabelle/HOL. The package supports partiality, pattern matching, mutual recursion, among others. It is realised as a definitional extension for Isabelle/HOL following the LCF tradition (recursive definitions are internally transformed into a non-recursive form, such that the function can be defined using standard definition facilities) [39].

Starting from the specification of a function the package inductively defines its graph and its domain, using the recursive structure of the definition. Then the graph is transformed into a total function modelling the specified function in the domain. The package then proves that the graph actually describes a function on the domain (i.e. function values always exist and are unique). The function package will also automatically derive an induction rule for the definition and prove that the definition is terminating on its domain.

2.4 Summary

In this Chapter we have surveyed literature we considered relevant to this thesis. After a brief description of different approaches to theory-exploration we have studied the systems AM, HR, MATHsAiD, MSC, IsaCoSy and Theorema. We have then described

the synthesis of conjectures and definitions using higher-order formulae (schemes) and outlined some examples. Lastly, we have described the Isabelle prover including its Pure and HOL logics. We have also briefly described Isabelle's tactical reasoning, the Simplifier, Isabelle's treatment of permutative rules and Isabelle's Function package.

Chapter 3

Background

In this chapter we aim to give a systematic presentation to background areas related to our work, namely terms of typed lambda calculus (section 3.1), substitution and rewriting of higher-order rewrite systems (section 3.2), termination and recursive path orders (section 3.3), the construction of convergent rewrite systems (section 3.4) and completion and proof by mathematical induction (section 3.5).

3.1 Terms of Typed Lambda Calculus

We mainly follow the notation in [44]. Given a finite set $\mathcal{S}$ of *type symbols*, and a denumerable set $\mathcal{S}^{\forall}$ of *type variables*, the set $\mathcal{T}_{\mathcal{S}^{\forall}}$ of *polymorphic types* is generated from these sets by the constructor $\rightarrow$ for *functional types*. The type constructor $\rightarrow$ associates to the right: read $\tau_1 \rightarrow \tau_2 \rightarrow \tau_3$ as $\tau_1 \rightarrow (\tau_2 \rightarrow \tau_3)$. Let τ be a type $\tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow \tau_0$ where $\tau_0 \in \mathcal{S}$ and $n \geq 0$. We will sometimes write $\overline{\tau_n} \rightarrow \tau_0$ for τ. A *signature* is a set of typed *function symbols* denoted by $\mathcal{F} = \bigcup_{\tau \in \mathcal{T}_{\mathcal{S}^{\forall}}} \mathcal{F}_\tau$. Terms are generated from a set of typed *variables* $\mathcal{V} = \bigcup_{\tau \in \mathcal{T}_{\mathcal{S}^{\forall}}} \mathcal{V}_\tau$ and a signature $\mathcal{F}$ by λ-abstraction and application and are denoted by $\mathcal{T}(\mathcal{F}, \mathcal{V})$. We write $t : \tau$ to indicate that the term t has type τ.

Typing rules restrict the set of terms $\mathcal{T}(\mathcal{F}, \mathcal{V})$ as follows:

$$\frac{x \in \mathcal{V}_\tau}{x : \tau} \qquad \frac{f \in \mathcal{F}_\tau}{f : \tau} \qquad \frac{s : \tau \rightarrow \tau' \qquad t : \tau}{(s\,t) : \tau'} \qquad \frac{x : \tau \qquad s : \tau'}{(\lambda x.\, s) : \tau \rightarrow \tau'}$$

In the sequel all λ-terms are assumed to be typed.

Instead of $\lambda x_1 \ldots \lambda x_n.\, s$ we also indulge with the notational convenience $\lambda x_1 \ldots x_n.\, s$ or just $\lambda \overline{x_n}.\, s$. Similarly instead of $(\ldots(t\, u_1)\ldots)\, u_n$ we use $t(u_1 \ldots u_n)$ or just $t(\overline{u_n})$.

We differentiate *free variables* from *bound variables* in that the latter are bound by λ-abstraction. The sets of function symbols, free variables and bound variables in a term t are denoted by $\mathcal{F}(t)$, $\mathcal{V}(t)$ and $\mathcal{B}(t)$, respectively. We say that a term t is closed if $\mathcal{V}(t) = \varnothing$. We assume the usual definition of α, β and η conversion between λ-terms. We follow the convention that terms which are α-congruent are identified (i.e. $\lambda x.x = \lambda y.y$). The β-*normal form* (η-*normal form*) of a term t is denoted as $t{\downarrow_\beta}$ ($t{\downarrow_\eta}$). Let t be in β-normal form; then t is of the form $\lambda\overline{x_k}.a(\overline{u_m})$ where a is called the *head* of t. The η-*expanded* form of t is defined by

$$t{\uparrow^\eta} = \lambda\overline{x_{n+k}}.\, a(\overline{u_m}{\uparrow^\eta}, x_{n+1}{\uparrow^\eta}, \ldots, x_{n+k}{\uparrow^\eta})$$

where $t : \overline{\tau_{n+k}} \to \tau$ and $x_{n+1}, \ldots, x_{n+k} \notin \mathcal{V}(\overline{u_m})$. We write $t{\updownarrow_\beta^\eta}$ instead of $t{\downarrow_\beta}{\uparrow^\eta}$. We say that a λ-term t is in *long* βη-*normal form* if $t = t{\updownarrow_\beta^\eta}$.

Example 1. *Assume* $G : (\tau_1 \to \tau_2) \to \tau_3$, $F : nat \to \tau \to \tau$, $rec : nat \to \tau \to (nat \to \tau \to \tau) \to \tau$, $suc : nat \to nat$, *and* $c : \tau_1 \to \tau_2$. *Some examples of terms and their* βη-*normal form are shown in the table 3.1:*

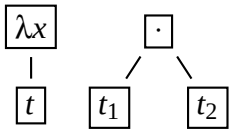
t	$t{\updownarrow_\beta^\eta}$
$\lambda x.\ c\ x$	$\lambda x.\ c\ x$
G	$\lambda x_1 : \tau_1 \to \tau_2.\, G\ (\lambda x_2 : \tau_1.\ x_1\ x_2)$
$G\ c$	$G\ (\lambda x_1 : \tau_1.\ c\ x_1)$
rec	$\lambda x_1, x_2, x_3.\ (rec\ x_1\ x_2\ (\lambda x_4, x_5.\ x_3\ x_4\ x_5))$
F	$\lambda x_1, x_2.\ (F\ x_1\ x_2)$
$rec\ (suc\ x)\ y\ F$	$rec\ (suc\ x)\ y\ (\lambda x_1, x_2.\ F\ x_1\ x_2)$
$F\ x\ (rec\ x\ y\ F)$	$F\ x\ (rec\ x\ y\ (\lambda x_1, x_2.\ F\ x_1\ x_2))$

Table 3.1: Some higher-order terms and their βη-normal form.

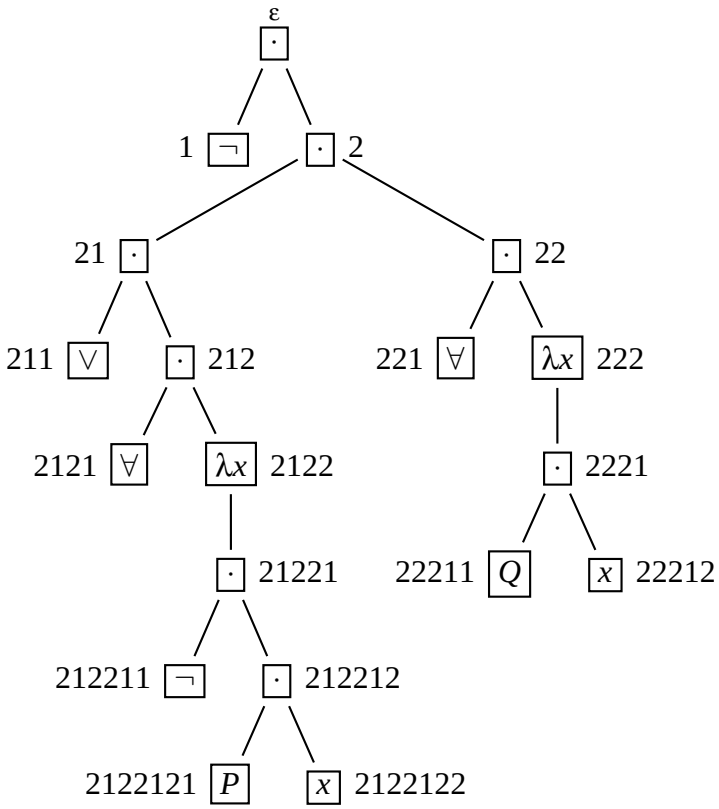
3.2 Substitution and Rewriting

There exist various formalisations of higher-order rewriting. In this thesis, we will consider higher-order rewrite systems (HRSs) as defined by Nipkow [44].

Higher-order terms can be viewed as trees by considering $\lambda x : \sigma.\ \_$ for each variable x and type σ, as a unary function symbol taking the term t as argument to construct the term $\lambda x : \sigma.t$. Abstraction and applications yield the following trees:



Positions are strings of positive integers. ε and $\cdot$ denote the empty string (root position) and string concatenation. $\mathcal{P}os(t)$ is the set of positions in t. The *subterm* of t at position p is denoted by $t|_p$. The result of replacing $t|_p$ at position p in t by u is written $t[u]_p$. For example, the following tree represents the term $\neg((\forall(\lambda x.\neg(P\ x)))\vee(\forall(\lambda x.(Q\ x))))$.



A *substitution* $\sigma : \mathcal{V} \to \mathcal{T}(\mathcal{F}, \mathcal{V})$ is a finite mapping from variables into terms of the same type. For $\sigma = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ we define $Dom(\sigma) = \{x_1, \ldots, x_n\}$ and $Cod(\sigma) = \{t_1, \ldots, t_n\}$. The application of a substitution to a term is defined by

$$\sigma(t) := (\lambda\overline{x_k}.t)(\overline{t_n})\downarrow^{\eta}_{\beta}$$

If there is a substitution σ such that $\sigma(s) = \sigma(t)$ we say s and t are *unifiable*. If there is a substitution σ such that $\sigma(s) = t$ we say that s *matches* the term t. The list of bound variables in a term t at position $p \in \mathcal{P}os(t)$ is denoted as

$$\begin{array}{lcl} \mathcal{B}(t,\varepsilon) & = & [\,] \\ \mathcal{B}((t_1\ t_2), i\cdot p) & = & \mathcal{B}(t_i, p) \\ \mathcal{B}(\lambda x.t, 1\cdot p) & = & x\cdot\mathcal{B}(t,p) \end{array}$$

A *renaming* ρ is an injective substitution with $Cod(\rho)\in\mathcal{V}$ and $Dom(\rho)\cap Cod(\rho)=\{\}$. An $\overline{x_k}-lifter$ of a term t away from W is a substitution $\sigma=\{F\to(\rho F)(\overline{x_k})|F\in\mathcal{V}(t)\}$ where ρ is a renaming with $Dom(\rho)=\mathcal{V}(t), Cod(\rho)\cap W=\{\}$ and $\rho F:\tau_1\to\ldots\tau_k\to\tau$ if $x_1:\tau_1,\ldots,x_k:\tau_k$ and $F:\tau$. An example taken from [44] is $\sigma=\{F\mapsto G\ x, S\mapsto T\ x\}$ which is an $x-lifter$ of $f\ (\lambda y.\ g\ (F\ y))\ S$ away from any W not containing G or T (the corresponding renaming is $\rho=\{F\mapsto G, S\mapsto T\}$).

Patterns are λ-terms in which the arguments of a free variable are (η-equivalent to) pairwise distinct bound variables. For instance, $(\lambda xyz.\ f\ (H\,x\,y)\ (H\,x\,z))$, $(\lambda x.\ c\,x)$ and $(\lambda x.\ F\ (\lambda z.\ x\ z))$ are patterns, while $(\lambda xy.\ G\,x\,x\,y)$, $(\lambda xy.\ F\ y\ c)$ and $(\lambda xy.\ H\ (F\ x)\ y)$ are not patterns.

A pair (l,r) of terms such that $l\notin\mathcal{V}$, l and r are of the same type and $\mathcal{V}(r)\subseteq\mathcal{V}(t)$ is called a *rewrite rule*. We write $l\to r$ for (l,r). A *higher-order rewrite system* (HRS for short) $\mathcal{R}$ is a set of rewrite rules. A set of rewrite rules whose left-hand sides are patterns is called a *pattern rewrite system* (PRS). The rewrite rules of a HRS $\mathcal{R}$ define a reduction relation $\to_{\mathcal{R}}$ on $\mathcal{T}(\mathcal{F},\mathcal{V})$ in the usual way.

$$s\to_{\mathcal{R}} t \Leftrightarrow \exists(l\to r)\in\mathcal{R}, p\in\mathcal{P}os(s), \sigma.\ s|_p=\sigma(l)\wedge t=s[\sigma(r)]_p$$

Example 2. *Let*

$$HRS=\{\neg(\neg P)\to P,\ \neg(P\vee Q)\to\neg P\wedge\neg Q,\ \neg(\forall(\lambda x.\ P\ x))\to\exists(\lambda x.\ \neg(P\ x))\}.$$

For readability we use $\forall x.\ P\ x$ and $\exists x.\ P\ x$ instead of $\forall(\lambda x.\ P\ x)$ and $\exists(\lambda x.\ P\ x)$. We also write $\vee$ and $\wedge$ as an infix. Then $\neg((\forall x.\ \neg(P\ x))\vee(\forall x.\ Q\ x))\to\neg(\forall x.\ \neg(P\ x))\wedge\neg(\forall x.\ Q\ x))\to(\exists x.\ \neg\neg(P\ x))\wedge\neg(\forall x.\ Q\ x))\to(\exists x.\ P\ x)\wedge\neg(\forall x.\ Q\ x))\to(\exists x.\ (P\ x))\wedge(\exists x.\ \neg(Q\ x))$, where the first reduction takes place at position $p_1=\varepsilon$ with the second identity and with the substitution $\sigma_1=\{P\mapsto\forall x.\neg(P\ x), Q\mapsto\forall x.Q\ x\}$, the second reduction takes place at position $p_2=1\cdot 2$ with the third identity and with the substitution $\sigma_2=\{P\mapsto\lambda x.\ \neg(P\ x)\}$, the third reduction takes place at position $p_3=1\cdot 2\cdot 2\cdot 1$ with the first identity and with the substitution $\sigma_3=\{P\mapsto P\ x\}$ and the last reduction takes place at position $p_4=2$ with the third identity and with the substitution $\sigma_4=\{P\mapsto\lambda x.\ Q\ x\}$. The reduction sequence is ilustrated in figure 3.1.

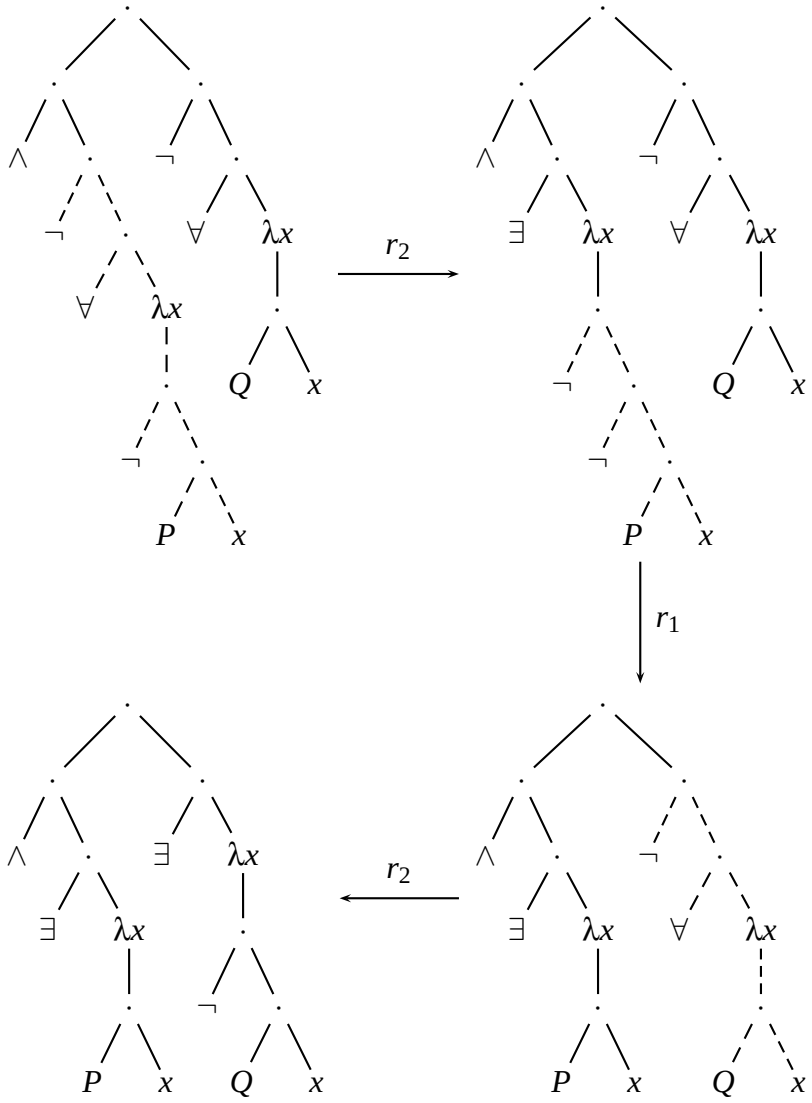


Figure 3.1: Reduction sequence over $\neg(\forall x.\neg P(x)) \wedge \neg(\forall x. Q(x)))$ with the rewrite rules $\neg(\neg P) \to P$ and $\neg(\forall(\lambda x. P(x))) \to \forall(\lambda x.\neg P(x))$ labeled r_1 and r_2 respectively. The dashed lines indicate the reduced expression within the term (also called redex).

The reflexive transitive closure of $\to_{\mathcal{R}}$ is denoted by $\overset{*}{\to}_{\mathcal{R}}$. A term t is reducible iff there is u such that $t \to_{\mathcal{R}} u$. A term u is a normal form of t iff $t \overset{*}{\to}_{\mathcal{R}} u$ and u is not reducible. A HRS is *terminating* if there exists no infinite reduction sequence $t_0 \to_{\mathcal{R}} t_1 \to_{\mathcal{R}} t_2 \ldots$.

3.3 Termination and Recursive Path Orders

An important property of term rewriting systems is termination. Termination plays a fundamental part in program verification, and is also used considerably in automated theorem proving. There are many techniques and methods to prove termination and describing them is far beyond the scope of this thesis. In this section we concentrate on an important approach based on recursive path orderings [21], generalised to the higher-order setting in [35] and recently improved in [37].

3.3.1 Polymorphic Higher-Order Recursive Path Order

A rewrite system is terminating if it does not admit infinite rewrite sequences. A common technique to prove termination of a rewrite system is to exhibit a *reduction order* $\succ$, which is a well-founded order on terms that is closed under contexts and substitution. Then if it can be proved that $\mathcal{R} \subseteq \succ$ then also the rewrite relation $\rightarrow_{\mathcal{R}} \subseteq \succ$, and hence $\mathcal{R}$ terminates. A *simplification order* is a reduction order that has the *subterm property*, i.e. for all terms $t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ and all positions $\mathcal{P}os(t) \setminus \{\varepsilon\}$ we have $t > t|_p$. A popular simplification order to prove termination of first-order rewrite systems is the recursive path order [21]. This ordering is defined in a recursive way by extending a well-founded ordering on function symbols. The obtained ordering is well founded on terms and is closed under contexts and substitutions. This yields the termination criterion: given a recursive path ordering $\succ$ show that $l \succ r$ for every rule $l \rightarrow r$.

A generalisation of the recursive path order to higher-order terms is described in [35] by Jouannaud and Rubio. This ordering is generated from two ingredients: a precedence on function symbols $>$ and a status for the function symbols. In the following, the notation $\overline{s}$ will be used to denote a list or a multiset, or a set of terms. The term $\Diamond(u, \overline{v})$ is called a (partial) *left-flattening*[1] of $s = u(v_1 \ldots v_n)$, u being possibly an application itself. If $>$ is a binary relation then the *lexicographic extension* of $>$ denoted as $(>)_{Lex}$ is defined as follows: $\{s_1, \ldots, s_n\} (>)_{Lex} \{t_1, \ldots, t_n\}$ if either $s_1 > t_1$ or $s_1 = t_1$ and $\{s_2, \ldots, s_n\} (>)_{Lex} \{t_2, \ldots, t_n\}$. The *multiset extension* of $>$, denoted by $(>)_{Mul}$, is defined as: $M (>)_{Mul} N$ if $M \neq N$ and $\forall n \in N - M. \exists m \in M - N. m > n$, with $-$ the difference on multisets (see [1] for details). We assume that every symbol $f \in \mathcal{F}$ has a status $stat_f \in \{Mul, Lex\}$ where f has multiset status if $f \in Mul$ and lexicographic status if $f \in Lex$.

The order is defined as follows:

[1] Which can be seen as the application operator written explicitly.

Definition 1. *Given two terms $s:\tau$ and $t:\tau$, the higher-order recursive path order $s \succ_{horpo} t$ iff one of the following conditions holds:*

1. $s := f(\overline{s})$ *with* $f \in \mathcal{F}$ *and* $u \succeq_{horpo} t$ *for some* $u \in \overline{s}$.
2. $s := f(\overline{s})$ *with* $f \in \mathcal{F}$ *and* $t = g(\overline{t})$ *with* $f >_{\mathcal{F}} g$ *and* A.
3. $s := f(\overline{s})$ *and* $t := g(\overline{t})$ *with* $f =_{\mathcal{F}} g \in Mul$ *and* $\overline{s}\left(\succ_{horpo}\right)_{Mul}\overline{t}$.
4. $s := f(\overline{s})$ *and* $t := g(\overline{t})$ *with* $f =_{\mathcal{F}} g \in Lex$ *and* $\overline{s}\left(\succ_{horpo}\right)_{Lex}\overline{t}$ *and* A.
5. $s := f(\overline{s})$ *with* $f \in \mathcal{F}$, $t := \Diamond(\overline{t})$ *is a partial left-flattening of* t, *and* A.
6. $s := s_1(s_2)$, $t := t_1(t_2)$ *and* $\{s_1, s_2\}\left(\succ_{horpo}\right)_{Mul}\{t_1, t_2\}$.
7. $s := \lambda x:\tau.u, t := \lambda x:\tau.v$ *and* $u \succ_{horpo} v$.

where $\succeq_{horpo}$ denotes the reflexive closure of $\succ_{horpo}$ and $A = \forall v \in \overline{t}.\, s \succ_{horpo} v$ or $u \succeq_{horpo} v$ for some $u \in \overline{s}$.

The first four clauses of the definition reduce directly to the usual recursive path order for first order terms, with the difference that instead of A for $\succ_{horpo}$ we have $s \succ v$ with $v \in \overline{t}$ for the first order case. This is not possible in the higher order case because the relation $\succ_{horpo}$ is only defined on terms of equal type. Other more advanced definitions of the higher-order recursive path order are given in [36, 37] and they can instead compare terms with different type.

Example 3. *Let $\mathcal{S} = \{nat\}$, $\mathcal{S}^{\forall} = \{\tau\}$ and $\mathcal{F} = \{0 : nat, suc : nat \to nat, rec : nat \to \tau \to (nat \to \tau \to \tau) \to \tau\}$. Gödel's recursor for natural numbers is defined by the following rewrite rules:*

$$\begin{aligned} rec(0, y, F) &= y \\ rec(suc(x), y, F) &= F(x, rec(x, y, F)) \end{aligned}$$

For the first rule we have $rec(0, y, F) \succ_{horpo} y$ by case 1. We apply case 5 for the second rule, and then show recursively the three remaining subgoals. First we have $F \succeq_{horpo} F$ trivially proved by reflexivity of $\succeq_{horpo}$. Second, we show $suc(x) \succ_{horpo} x$ by case 1. Third, we have $rec(suc(x), y, F) \succ_{horpo} rec(x, y, F)$ proved by case 3 calling again recursively for $suc(x) \succ_{horpo} x$ (example taken from [35]).

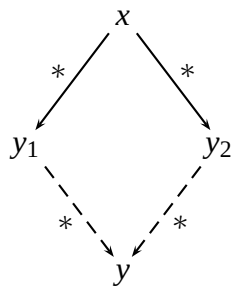
Example 4. *Let* $\mathcal{S} = \{nat\}, \mathcal{S}^{\forall} = \varnothing$ *and* $\mathcal{F} = \{0 : nat, suc : nat \to nat, f : nat \to nat \to nat \to nat\}$. *The rewrite rule:*

$$f(0, suc(0), x) \quad = \quad f(x, x, x)$$

cannot be shown to be terminating using the higher-order recursive path order. The reason is that clauses 3 and 4 fail to yield that $f(0, suc(0), x) \succ_{horpo} f(x, x, x)$ *because neither* $\{0, suc(0), x\} \left(\succ_{horpo}\right)_{Mul} \{x, x, x\}$ *nor* $\{0, suc(0), x\} \left(\succ_{horpo}\right)_{Lex} \{x, x, x\}$ *succeed.*

3.4 Confluence and Completion

An important question in term rewriting is whether the result of a rewrite sequence, if it exists, is unique. We say that an expresion x is in normal form if there is no y such that $x \to y$. Uniqueness of normal forms is guaranteed by a property in rewriting called *confluence*. For every y_1 and y_2 such that $x \xrightarrow{*} y_1$ and $x \xrightarrow{*} y_2$ there is a y such that $y_1 \xrightarrow{*} y$ and $y_2 \xrightarrow{*} y$. We depict this in the following picture where solid lines stand for universal quantification and dashed lines for existential quantification:



In general, the problem of deciding whether a rewrite system is confluent is undecidable. However, confluence is decidable for terminating finite rewrite systems by checking for the so-called critical pairs. In [44], Tobias Nipkow defines critical pairs for higher order rewrite systems as:

Definition 2. *Let* $l_1 \to r_1$ *and* $l_2 \to r_2$ *be two rewrite rules in a PRS and* $p \in \mathcal{P}os(l_1)$ *such that:*

- $\mathcal{V}(l_1) \cup \mathcal{B}(l_1) = \{\}$,
- *the head of* $l_1|_p$ *is not a free variable in* l_1, *and*
- *the two patterns* $\lambda\overline{x_k}.(l_1|_p)$ *and* $\lambda\overline{x_k}.(\sigma(l_2))$ *where* $\{\overline{x_k}\} = \mathcal{B}(l_1, p)$ *and* σ *is an* $\overline{x_k} - lifter$ *of* l_2 *away from* $\mathcal{V}(l_1)$, *have a most general unifier* Θ.

Then the pattern l_1 overlaps *the pattern* l_2 *at position* p. *The rewrite rules yield the critical pair* $< \theta(r_1), \theta(l_1[\sigma(r_2)]_p) >$.

Example 5. *Let* $\mathcal{S} = \{bool\}$, $\mathcal{S}^{\forall} = \{\tau\}$ *and* $\mathcal{F} = \{\neg : bool \to bool, \vee, \wedge : bool \to bool \to bool, \forall, \exists : (\tau \to bool) \to bool\}$. *Again, for readability we use* $\forall x.\ P\ x$ *and* $\exists x.\ P\ x$ *instead of* $\forall(\lambda x.\ P\ x)$ *and* $\exists(\lambda x.\ P\ x)$. *We also write each of* $\vee$ *and* $\wedge$ *as an infix. The following terminating rewrite system*

$$\begin{aligned} \neg(\neg P) &= P \\ \neg(P \wedge Q) &= (\neg P) \vee (\neg Q) \\ \neg(P \vee Q) &= (\neg P) \wedge (\neg Q) \\ \neg(\forall x.\ P\ x) &= \exists x.\ \neg(P\ x) \\ \neg(\exists x.\ P\ x) &= \forall x.\ \neg(P\ x) \end{aligned}$$

produces 5 critical pairs obtained through unification of the left-hand side of some rule with the subterm $\neg P$ *of* $\neg(\neg P)$.

$$\begin{gathered} < \neg P, \neg P > \\ < P \wedge Q, \neg((\neg P) \vee (\neg Q)) > \\ < P \vee Q, \neg((\neg P) \wedge (\neg Q)) > \\ < \forall x. P(x), \neg(\exists x. \neg P(x)) > \\ < \exists x. P(x), \neg(\forall x. \neg P(x)) > \end{gathered}$$

The first pair is trivially joinable. The second is joinable because $\neg((\neg P) \vee (\neg Q)) \to (\neg(\neg P)) \wedge (\neg(\neg Q)) \to P \wedge Q$. *The third is joinable because* $\neg((\neg P) \wedge (\neg Q)) \to (\neg(\neg P)) \vee (\neg(\neg Q)) \to P \vee Q$. *The fourth pair is joinable because* $\neg(\exists x. \neg P(x)) \to \forall x. \neg(\neg P(x)) \to \forall x. P(x)$. *The fifth pair is joinable because* $\neg(\forall x. \neg P(x)) \to \forall x. \neg(\neg P(x)) \to \forall x. P(x)$. *Hence, the system is confluent and produces unique normal forms.*

The process of transforming a finite set of equations into a terminating and confluent rewrite system is called *completion*. A completion algorithm usually takes as input a set of identities E and a reduction ordering $\succ$ and attempts to produce a set of rules R such that R is *convergent* (terminating and confluent) and equivalent to E (i.e. with the same equational theory $\overset{*}{\leftrightarrow}_E = \overset{*}{\leftrightarrow}_R$). Completion is typically described as a collection of inference rules, which are given in table 3.2, that work on pairs (E, R) where E is a finite set of identities and R a finite set of rewrite rules. The idea is to transform an initial pair $(E, \varnothing)$ into the pair $(\varnothing, R)$ such that R is convergent and equivalent to E [1, 2]. Here $s \doteq t \in E$ should be seen as $s = t \in E \ \vee\ t = s \in E$ and the symbol $\rhd$ is defined as: $s \rhd l$ if and only if some subterm of s is an instance of l but not vice versa.

We use the notation $(E, R) \vdash_C (E', R')$ to indicate that (E, R) can be transformed to (E', R') by any of the rules in table 3.2.

Orientation:	$\frac{(E \cup \{s \doteq t\}, R)}{(E, R \cup \{s \to t\})}$	if $s \succ t$
Deduction:	$\frac{(E, R)}{(E \cup \{s = t\}, R)}$	if $s \leftarrow_R u \to_R t$
Deletion:	$\frac{(E \cup \{s = s\}, R)}{(E, R)}$	
Simplification:	$\frac{(E \cup \{s \doteq t\}, R)}{(E \cup \{u \doteq t\}, R)}$	if $s \to_R u$
Composition:	$\frac{(E, R \cup \{s \to t\})}{(E, R \cup \{s \to u\})}$	if $t \to_R u$
Collapse:	$\frac{(E, R \cup \{s \to t\})}{(E \cup \{v = t\}, R)}$	if $s \to_R v$ and $s \rhd v$

Table 3.2: The inference rules for completion.

Orientation makes use of $\succ$ to orient the equalities in E and adds the corresponding rule to R ($\doteq$ is used to avoid having two versions of the rule). A practical implementation of *Deduction* is to add critical pairs in R to E. *Deletion* removes trivial identities from E. *Simplification* simplifies identities in E w.r.t. R. Reduction of right-hand sides of rules is performed by *Composition*. $t \to u$ is kept as a rule because $s \to_R t \to_R u$ implies $s \succ t \succ u$, assuming that termination of R can be shown by $\succ$. Reduction of left-hand sides of rules is performed by *Collapse*.

Example 6. *Consider the theory where* $E_0 := \{(x*y)*(y*z) = y\}$ *and let* $\succ$ *be an arbitrary simplification order. Because of the subterm property of* $\succ$*, we have* $(x*y)*(y*z) \succ y$*. Hence an application of Orientation yields* $(E_0, \varnothing) \vdash_C (\varnothing, \{(x*y)*(y*z) \to y\})$*. Two applications of Deduction produce the pair:*

$$\left(\left\{ \begin{array}{l} y*z = y*((y*z)*x), \\ x*y = (z*(x*y))*y \end{array} \right\}, \{(x*y)*(y*z) \to y\} \right)$$

This is because the rule $(x*y)*(y*z) \to y$ *has two critical pairs when it is overlapped with its renamed copy* $(x'*y')*(y'*z') \to y'$*. The critical pairs emerge from the unification of* $(x*y)*(y*z)$ *and the subterms* $x'*y'$ *and* $y'*z'$ *of the renamed rule.*

$$< y*z, y*((y*z)*z') >$$
$$< x*y, (x'*(x*y))*y >$$

Two applications of Orientation turn the previous pair into the new pair (E_1, R_1)*:*

$$\left(\varnothing, \left\{ \begin{array}{lcl} (x*y)*(y*z) & \to & y, \\ y*((y*z)*x) & \to & y*z, \\ (z*(x*y))*y & \to & x*y \end{array} \right\}\right)$$

Again note that $y*((y*z)*x) \succ y*z$ *and* $(z*(x*y))*y \succ x*y$ *because of the subterm property. Deduction places the critical pairs between these rules in the first component of the pair and Simplification reduces these critical pairs to trivial identities. All trivial identities are then eliminated by Deletion and the completion procedure terminates with success.*

Completion does not always succeed on a set of identities E and a reduction ordering $\succ$, i.e. terminate with the pair $(\varnothing, R)$ where R is convergent and equivalent to E. Failure occurs when an initial identity or a normal form of a critical pair can not be oriented by the given ordering $\succ$. An example of this failure is the theory $E := \{x*(y+z) = (x*y)+(x*z), (u+v)*w = (u*w)+(v*w)\}$ and the higher-order recursive path ordering $\succ_{horpo}$ induced by the precedence $* > +$. Two applications of Orientation yield the pair

$$\left(\varnothing, \left\{ \begin{array}{lcl} x*(y+z) & \to & (x*y)+(x*z), \\ (u+v)*w & \to & (u*w)+(v*w) \end{array} \right\}\right)$$

which after overlapping the first rule and the second rule by Deduction produce the critical pair:

$$< (u+v)*y+(u+v)*z, u*(y+z)+v*(y+z) > .$$

This identity is normalised by Simplification producing a non-orientable identity (cannot be ordered).

Completion can also fail in an infinite execution of the rules in table 3.2. An example of non-termination of completion is described with the theory $E_0 = \{f\ (g\ (f\ x)) = f\ (g\ x)\}$ and the higher-order recursive path order $\succ$ induced by the precedence $g > h > f$. Here an application of Orientation yields the pair:

$$\left(\varnothing, \left\{ \begin{array}{lcl} f\ (g\ (f\ x)) & \to & f\ (g\ x) \end{array} \right\}\right),$$

which after overlapping the rule with its renamed copy yields the critical pair

$$< f\ (g\ (f\ (g\ x))),\ f\ (g\ (g\ (f\ x))) >$$

in an application of Deduction producing the pair

$$\left(\left\{\ f\ (g\ (f\ (g\ x))) = f\ (g\ (g\ (f\ x)))\ \right\}, \left\{\ f\ (g\ (f\ x))\ \ \rightarrow\ \ f\ (g\ x)\ \right\}\right)$$

which by Simplification yields:

$$\left(\left\{\ f\ (g\ (g\ x)) = f\ (g\ (g\ (f\ x)))\ \right\}, \left\{\ f\ (g\ (f\ x))\ \ \rightarrow\ \ f\ (g\ x)\ \right\}\right).$$

Orientation then generates the pair

$$\left(\varnothing, \left\{ \begin{array}{rcl} f\ (g\ (f\ x)) & \rightarrow & f\ (g\ x) \\ f\ (g\ (g\ (f\ x))) & \rightarrow & f\ (g\ (g\ x)) \end{array} \right\}\right);$$

a new application of Deduction, Simplification and Orientation produces the pair

$$\left(\varnothing, \left\{ \begin{array}{rcl} f\ (g\ (f\ x)) & \rightarrow & f\ (g\ x) \\ f\ (g\ (g\ (f\ x))) & \rightarrow & f\ (g\ (g\ x)) \\ f\ (g\ (g\ (g\ (f\ x)))) & \rightarrow & f\ (g\ (g\ (g\ x))) \end{array} \right\}\right),$$

and this process continues generating always a pair of the form:

$$\left(\varnothing, \left\{ \begin{array}{rcl} f\ (g\ (f\ x)) & \rightarrow & f\ (g\ x) \\ & \vdots & \\ f\ (g^n\ (f\ x)) & \rightarrow & f\ (g^n\ x) \end{array} \right\}\right).$$

3.5 Completion as a Source for Invention

One practical benefit of the completion approach is that it allows theory-exploration. In many applications of automated deduction such as software and hardware verification, various theorems can be routinely proved using the same basic theory. Since the completion procedure works only in the theory, independent of the goal, the resulting rewrite system generated by the completion process can be reused for proving other goals. The completion process often creates useful lemmas or generalisations as new critical pairs of the rewrite system (see section 4.8). Completion is often thought of as a ‘compilation’ phase because the given axioms are ‘compiled’ into a confluent set of rewrite rules and this makes the given theory more efficient for theorem proving (‘execution’ phase). The lemma generation power of the completion process is illustrated in the following example. Consider the theory

$$\mathcal{E}_1 := \left\{ \begin{array}{rcl} 0+y & = & y \\ (suc\ x)+y & = & suc\ (x+y) \\ (x+y)+z & = & x+(y+z) \\ x+0 & = & x \\ x+(suc\ 0) & = & suc\ x \end{array} \right\}.$$

In this theory, five applications of Orientation yield the pair

$$\left(\varnothing, \left\{ \begin{array}{rcl} 0+y & \to & y \\ (suc\ x)+y & \to & suc\ (x+y) \\ (x+y)+z & \to & x+(y+z) \\ x+0 & \to & x \\ x+(suc\ 0) & \to & suc\ x \end{array} \right\} \right).$$

An application of Deduction produce the pair

$$\left(\left\{ x+(y+suc\ 0) \ = \ suc\ (x+y) \right\}, \left\{ \begin{array}{rcl} 0+y & \to & y \\ (suc\ x)+y & \to & suc\ (x+y) \\ (x+y)+z & \to & x+(y+z) \\ x+0 & \to & x \\ x+(suc\ 0) & \to & suc\ x \end{array} \right\} \right).$$

This is because the rules $(x+y)+z = x+(y+z)$ and $x+(suc\ 0) = suc\ x$ have the critical pair $x+(y+suc\ 0) = suc\ (x+y)$. An application of Simplification followed by Orientation yield

$$\left(\varnothing, \left\{ \begin{array}{rcl} 0+y & \to & y \\ (suc\ x)+y & \to & suc\ (x+y) \\ (x+y)+z & \to & x+(y+z) \\ x+0 & \to & x \\ x+(suc\ 0) & \to & suc\ x \\ x+(suc\ y) & \to & suc\ (x+y) \end{array} \right\} \right).$$

Finally an application of Collapse, Simplification and Deletion (with $x+(suc\ 0) \to suc\ x$) produce the pair

$$\left(\varnothing, \left\{\begin{array}{rcl} 0+y & \to & y \\ (suc\ x)+y & \to & suc\ (x+y) \\ (x+y)+z & \to & x+(y+z) \\ x+0 & \to & x \\ x+(suc\ y) & \to & suc\ (x+y) \end{array}\right\}\right).$$

Here we can see that the equation $x+(suc\ 0) = suc\ x$ is generalised to $x+(suc\ y) = suc\ (x+y)$ by the completion process (see example 12 in section 4.5.2.4). The power of completion also apply to higher-order theories. Consider for example, the isomorphic theory (because $\oplus = +$)

$$\mathcal{E}_2 := \left\{\begin{array}{rcl} rec\ 0\ y\ z & = & y \\ rec\ (suc\ x)\ y\ z & = & z\ x\ (rec\ x\ y\ z) \\ \oplus & = & (\lambda xy.\ rec\ x\ y\ (\lambda uv.\ suc\ v)) \\ rec\ (rec\ x\ y\ (\lambda u.\ suc))\ z\ (\lambda u.\ suc) & = & rec\ x\ (rec\ y\ z\ (\lambda u.\ suc))\ (\lambda u.\ suc) \\ rec\ x\ 0\ (\lambda u.\ suc) & = & x \\ rec\ x\ (suc\ 0)\ (\lambda u.\ suc) & = & suc\ x \end{array}\right\}.$$

Here a similar generalisation process occur but now with higher-order equations. In this case the equation $rec\ x\ (suc\ 0)\ (\lambda u.\ suc) = suc\ x$ is generalised to $rec\ x\ (suc\ y)\ (\lambda u.\ suc) = suc\ (rec\ x\ y\ (\lambda u.\ suc))$ (see example 13 in appendix B).

3.6 Summary

This chapter presents some background material important for the development of this thesis. We first described the terms of type lambda calculus followed by a brief introduction to Nipkow's higher-order rewrite systems (HRS). We then studied termination and higher-order recursive path orders. Finally, we presented convergence of rewrite systems followed by a description of an algorithm which transforms a set of equations into a convergent rewrite system called completion and its application to mathematical theory-exploration.

Chapter 4

A framework for Schematic Discovery

This chapter describes the techniques used to generate conjectures and definitions using higher-order formulae in IsaScheme. We also give a first general view of how IsaScheme works and the various components it depends on, both internally to IsaScheme and externally. An information flow view of IsaScheme is depicted in figure 4.1. In this view, the user is assumed to supply IsaScheme with a set of schemes, a set of closed terms and a tactic to discharge the proof obligations. The set of closed terms eventually become just the function symbols (or any other term of interest of the user) that the user wish to find properties about (see section 4.6).

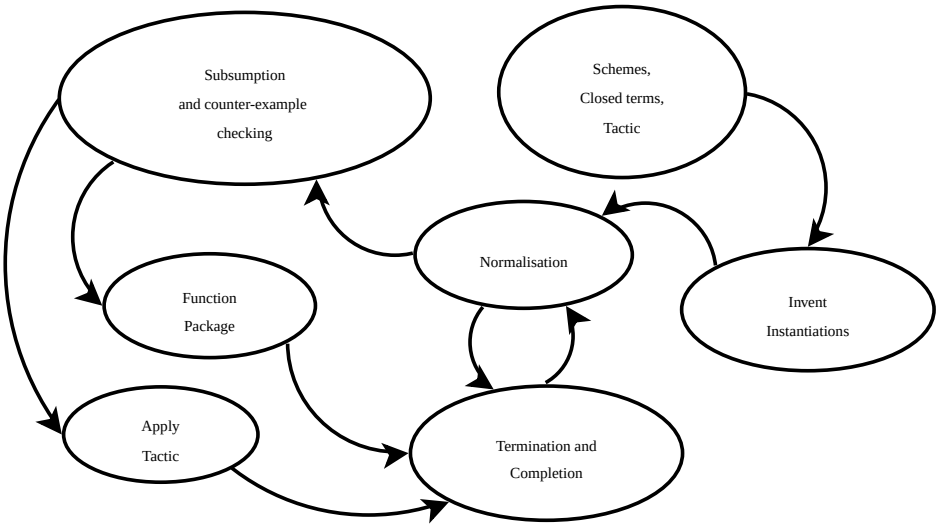


Figure 4.1: Information flow between components of IsaScheme.

Firstly, we describe how HOL can be used to encode mathematical knowledge in section 4.1. This knowledge is then exploited during the synthesis of conjectures and definitions (discussed in section 4.2). Here we describe how we can easily automate

the instantiation process of schemes on top of the simply-typed lambda calculus of Isabelle/HOL [55]. This process will often produce equivalent instantiations leading to redundancies during the exploration of the theory. We show how new definitions and the lemmata discovered during the exploration of the theory can be used not only to strengthen Isabelle's tools, such as the *Simplifier* [56] or the *function package* [39], but also to reduce redundancies inherent in most theory-formation systems (sections 4.3 and 4.4). We also outline the algorithms for theory-exploration where the processes of theorem and definition discovery are linked together in section 4.5. The algorithms outlined receive as input a set of closed terms in the theory being explored. An automatic generation of such closed terms is considered in section 4.6. We finally remark on the applicability of the developed model and discuss its ability to create higher-order conjectures and definitions in 4.7. We conclude with a summary in 4.9.

4.1 Representation of Schemes

A *scheme* is a higher-order formula intended to generate new *definitions* of the underlying theory and *conjectures* about them. However, not every higher-order formula is a scheme. Here, we formally define schemes.

Definition 3. *A* **scheme** *s is a (non-recursive) constant definition of a proposition in HOL,* $s_n \equiv \lambda\overline{x}.\, t$*, which we write in the form* $s_n\ \overline{x} \equiv t$*.*

For the scheme $s_n\ \overline{x} \equiv t$, $\overline{x}$ are free variables and $t : bool$ does not contain s_n, does not refer to undefined symbols and does not introduce extra free variables. The scheme (where *dvd* means "divides") $prime\ P \equiv 1 < P \wedge ((dvd\ M\ P) \rightarrow M = 1 \vee M = P)$ is flawed because it introduces the extra free variable M on the right hand side. The correct version is $prime\ P \equiv 1 < p \wedge (\forall m.\ (dvd\ m\ P) \rightarrow m = 1 \vee m = P)$ assuming that all symbols are properly defined.

Definition 4. *We say that a scheme* $s := s_n\ \overline{x} \equiv t$ *is a* **definitional scheme** *if t has the form* $\exists\overline{f}\ \forall\overline{y} \wedge_{i=1}^{m} l_i = r_i$ *and the head of each* $l_i \in \overline{f}$*. The* **defining equations** *of s are then denoted by* $l_1 = r_1, \ldots, l_m = r_m$*.*

Examples of valid schemes are listed below.

$$\begin{aligned} true &\equiv \top \\ comm\ P &\equiv \forall xy.\ P\ x\ y = P\ y\ x \\ assoc\ P &\equiv \forall xyz.\ P\ (P\ x\ y)\ z = P\ x\ (P\ y\ z). \end{aligned}$$

The following are examples of valid definitional schemes.

$$\left(\begin{array}{l} \mathit{def}\text{-}\mathit{scheme}\ G\ H\ I\ J \equiv \\ \qquad \exists f.\ \forall x y z.\ \ \bigwedge \left\{ \begin{array}{l} f\ G\ y\ = y \\ f\ (H\ z\ x)\ y = I\ (J\ z\ y)\ (f\ x\ y) \end{array} \right. \end{array}\right) \tag{4.1}$$

$$\left(\begin{array}{l} \mathit{mutual}\text{-}\mathit{def}\text{-}\mathit{scheme}\ G\ H\ I\ J\ K\ L \equiv \\ \qquad \exists f_1\ f_2.\ \forall x y.\ \ \bigwedge \left\{ \begin{array}{l} f_1\ G = H \\ f_2\ G = I \\ f_1\ (J\ z\ x) = K\ z\ (f_2\ x) \\ f_2\ (J\ z\ x) = L\ z\ (f_1\ x) \end{array} \right. \end{array}\right) \tag{4.2}$$

The definitional scheme (4.2) captures the idea of two mutual functions defined recursively. Here the existentially quantified variables (f in scheme (4.1) and f_1 and f_2 in scheme (4.2)) stand for the new functions to be defined.

Invalid definitional schemes are listed below.

$$\left(\begin{array}{l} \mathit{def}\text{-}\mathit{scheme}\ G\ H\ I\ J \equiv \\ \qquad \exists f.\ \forall x y z.\ \ \bigwedge \left\{ \begin{array}{l} y = f\ G\ y \\ f\ (H\ z\ x)\ y = I\ (J\ z\ y)\ (f\ x\ y) \end{array} \right. \end{array}\right) \tag{4.3}$$

$$\left(\begin{array}{l} \mathit{mutual}\text{-}\mathit{def}\text{-}\mathit{scheme}\ G\ H\ I\ J\ K\ L \equiv \\ \qquad \forall x y.\ \exists f_1\ f_2.\ \ \bigwedge \left\{ \begin{array}{l} f_1\ G = H \\ f_2\ G = I \\ f_1\ (J\ z\ x) = K\ z\ (f_2\ x) \\ f_2\ (J\ z\ x) = L\ z\ (f_1\ x) \end{array} \right. \end{array}\right) \tag{4.4}$$

The scheme 4.3 is not a definitional scheme because the head of the left-hand side of the equation $y = f(G,y)$ is not f. The scheme 4.4 is not a definitional scheme because the quantifiers occur in the wrong order.

4.2 Generation of Instantiations

In this section we describe the technique used to instantiate schemes automatically. In order to perform this instantiation automatically, we first need to decide which terms will be used during the instantiation process. For example, suppose we want to explore properties about addition of natural numbers ($+$) and about the function length of lists (*len*). The function symbols might be defined in different Isabelle theories and other irrelevant definitions, which we wish to ignore, could be present. This situation suggests

the use of a set of terms, as an argument to the system, to concentrate the exploration process around a specific domain of the theories. Here we define some preliminary concepts considering this specification.

Definition 5. *For a scheme $s := u \equiv v$, the set of* **schematic substitutions** *with respect to a (finite) set of closed terms $X \subset \mathcal{T}(\mathcal{F}, \mathcal{V})$ is defined by:*

$$Sub(s,X) := \{\sigma \mid \sigma(v) \text{ is closed } \wedge\, Cod(\sigma) \subseteq X)\}.$$

Ensuring that $\sigma(v)$ is a closed term helps avoid overgeneralisations of conjectures or definitions, e.g. it is impossible to prove $\forall xyz.\ x * P(y,z) = P(x*y, x*z)$ where P is free. Definition 5 also restricts substitutions to be within X. The problem of finding the substitutions $Sub(s,X)$ of a scheme s given a set of terms X can be solved as follows. The free variables $\mathcal{V}(s) = \{v_1, \ldots, v_n\}$ in the scheme are associated with their initial domain $D_{0i} = \{x \in X \mid v_i \text{ and } x \text{ are unifiable}\}$ for $1 \leq i \leq n$. The typing information of the partially instantiated scheme is the only constraint during the instantiation of variables. Each time a variable v_i is instantiated to $x \in D_{ki}$ the domains $D_{(k+1)j}$ for $i < j \leq n$ of the remaining variables must be updated w.r.t the most general unifier σ_{mgu} of v_i and x. Variables are instantiated sequentially and if a partial instantiation leaves no possible values for a variable then backtracking is performed to the most recently instantiated variable that still has alternatives available. This process is repeated using backtracking to exhaust all possible schematic substitutions. Note that any constraint-satisfaction algorithm is compatible to our problem of finding the schematic substitutions.

Theorem 1. *Given a term t and a finite set of closed terms X then the number of schematic substitutions is bounded by $O(|X|^{|\mathcal{V}(t)|})$.*

Proof. Since the number of free variables in t is $|\mathcal{V}(t)|$ and in the worst case each of them can be instantiated to every term $x \in X$, we have

$$\underbrace{|X| \cdots |X|}_{|\mathcal{V}(t)| \text{ times}} \;=\; |X|^{|\mathcal{V}(t)|}.$$

□

Termination of the algorithm follows from the fact that it applies simple depth-first search on a finite search space and thus, infinite paths do not exists. Furthermore, the sequential instantiation of variables avoids cycles in the search and thus, it terminates.

Example 7. *Let $\mathcal{F}$ be a signature consisting of $\mathcal{F} := \{+ : nat \to nat \to nat, * : nat \to nat \to nat, @ : \tau\ list \to \tau\ list \to \tau\ list, map : (\tau \to \rho) \to \tau\ list \to \rho\ list\}$. Also let $X = \{+, *, @, map\}$ and s be the following scheme (here we assume the most general type inferred for the scheme).*

$$\left(\begin{array}{l} left\text{-}distributivity\ P\ Q \equiv \\ \qquad \forall xyz.\ Q\ x\ (P\ y\ z) = P\ (Q\ x\ y)\ (Q\ x\ z) \end{array} \right) \tag{4.5}$$

Figure (4.2) illustrates how $Sub(s,X)$ is evaluated following a sequential instantiation of the free variables of s.

For a scheme s, the generated schematic substitutions are used to produce instantiations of s; i.e. conjectures or definitions.

Definition 6. *Given $\sigma \in Sub(s,X)$, the* **instantiation of the scheme** *$s := u \equiv v$ with σ is defined by*

$$inst(u \equiv v, \sigma) := \sigma(v).$$

Definition 7. *For a scheme s, the* **set of instantiations** *$Insts(s,X)$ with respect to a (finite) set of closed terms $X \subset \mathcal{T}(\mathcal{F}, \mathcal{V})$ is denoted by*

$$Insts(s,X) := \{inst(s,\sigma) \mid \sigma \in Sub(s,X)\}. \tag{4.6}$$

Example 8. *The instantiations generated from scheme (4.5) and the set of terms $X = \{+, *, @, map\}$ are depicted in table 4.1.*

$Sub(s,X)$	$Insts(s,X)$
$\sigma_1 = \{P \mapsto +, Q \mapsto +\}$	$\forall xyz.\ x+(y+z) = (x+y)+(x+z)$
$\sigma_2 = \{P \mapsto +, Q \mapsto *\}$	$\forall xyz.\ x*(y+z) = x*y+x*z$
$\sigma_3 = \{P \mapsto *, Q \mapsto +\}$	$\forall xyz.\ x+y*z = (x+y)*(x+z)$
$\sigma_4 = \{P \mapsto *, Q \mapsto *\}$	$\forall xyz.\ x*(y*z) = (x*y)*(x*z)$
$\sigma_5 = \{P \mapsto @, Q \mapsto @\}$	$\forall xyz.\ x\ @\ (y\ @\ z) = (x\ @\ y)\ @\ (x\ @\ z)$
$\sigma_6 = \{P \mapsto @, Q \mapsto map\}$	$\forall xyz.\ map\ x\ (y\ @\ z) = (map\ x\ y)\ @\ (map\ x\ z)$

Table 4.1: The instantiations induced by the schematic substitutions of example 7.

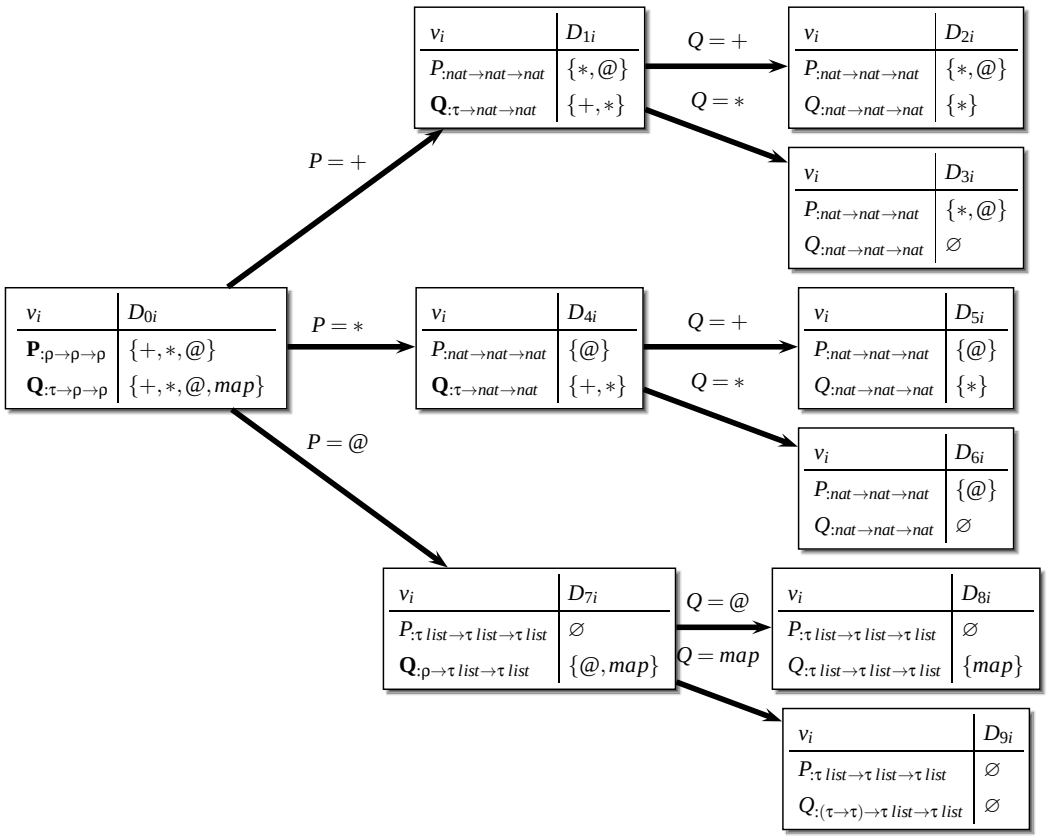


Figure 4.2: Sequential evaluation of $Sub(s,X)$ where s is the scheme (4.5) and $X = \{+ : nat \to nat \to nat, * : nat \to nat \to nat, @ : \tau\ list \to \tau\ list \to \tau\ list, map : (\tau \to \rho) \to \tau\ list \to \rho\ list\}$. Each box shows the unified and not-unified (in bold) variables and their domain during the evaluation. The output of the algorithm is the set of substitutions $\{\sigma_1 = \{P \mapsto +, Q \mapsto +\}, \sigma_2 = \{P \mapsto +, Q \mapsto *\}, \sigma_3 = \{P \mapsto *, Q \mapsto +\}, \sigma_4 = \{P \mapsto *, Q \mapsto *\}, \sigma_5 = \{P \mapsto @, Q \mapsto @\}, \sigma_6 = \{P \mapsto @, Q \mapsto map\}\}$. Note that a unified variable potentially changes the types of the rest of the variables, restricting their domain.

4.3 Identification of Equivalent Instantiations

Processing the instantiations (conjectures and definitions) of a scheme could be a demanding task. In the worst case, the number of substitutions $\sigma : V \to X$ is $|X|^{|V|}$. However, we can reduce the number of conjectures and definitions by noticing that two different substitutions σ_1 and σ_2 could lead to equivalent instantiations.

Table 4.2 shows a sampling of the set of instantiations $Insts(s,X)$ obtained from

the set of closed terms $X := \{0, +, (\lambda x.\ x)\}$ and the following definitional scheme.

$$\left(\begin{array}{l} def\text{-}scheme\ G\ H\ I \equiv \\ \qquad \exists f.\ \forall xy.\quad \bigwedge \left\{ \begin{array}{l} f\ G\ y = H\ G\ G \\ f\ (suc\ x)\ y = I\ y\ (f\ x\ y) \end{array} \right. \end{array} \right) \tag{4.7}$$

In Table 4.2 $inst(s, \sigma_{N1})$ and $inst(s, \sigma_{N2})$ are clearly equivalent[1]. The key ingredient for automatically detecting equivalent instantiations is a term rewrite system (HRS) $\mathcal{R}$. This rewrite system is used to normalise terms such that only irreducible terms are considered [33].

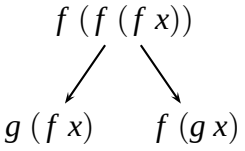
$Sub(s,X)$	$Insts(s,X)$
$\sigma_{N1} = \left\{ \begin{array}{c} G \mapsto 0, H \mapsto + \\ I \mapsto + \end{array} \right\}$	$\exists f.\ \forall xy. \bigwedge \left\{ \begin{array}{l} f\ 0\ y = 0 + 0 \\ f\ (suc\ x)\ y = y + (f\ x\ y) \end{array} \right.$
$\sigma_{N2} = \left\{ \begin{array}{c} G \mapsto 0, H \mapsto (\lambda xy.\ x) \\ I \mapsto + \end{array} \right\}$	$\exists f.\ \forall xy. \bigwedge \left\{ \begin{array}{l} f\ 0\ y = 0 \\ f\ (suc\ x)\ y = y + (f\ x\ y) \end{array} \right.$
$\sigma_{N3} = \left\{ \begin{array}{c} G \mapsto 0, H \mapsto + \\ I \mapsto (\lambda xy.\ x) \end{array} \right\}$	$\exists f.\ \forall xy. \bigwedge \left\{ \begin{array}{l} f\ 0\ y = 0 + 0 \\ f\ (suc\ x)\ y = y \end{array} \right.$
$\sigma_{N4} = \left\{ \begin{array}{c} G \mapsto 0, H \mapsto (\lambda xy.\ x) \\ I \mapsto (\lambda xy.\ x) \end{array} \right\}$	$\exists f.\ \forall xy. \bigwedge \left\{ \begin{array}{l} f\ 0\ y = 0 \\ f\ (suc\ x)\ y = y \end{array} \right.$

Table 4.2: Redundant definitions generated from the definitional scheme 4.7. Note that the instantiations $inst(s, \sigma_{N1})$ and $inst(s, \sigma_{N2})$ are equivalent as $0 + 0$ can be 'reduced' (within the theory) to 0. $inst(s, \sigma_{N3})$ and $inst(s, \sigma_{N4})$ are similarly equivalent.

An easy and practical way to find normal forms is to use a HRS $\mathcal{R}$ with the property of being *terminating*. For a finite terminating rewrite system, a normal form of a given term can be found by simple depth-first search. All functions in Isabelle/HOL are terminating to prevent inconsistencies [39]. Even partially defined functions need to be shown to be terminating in their partial domain. Therefore, the defining equations for a newly introduced function symbol can be used as a normalising HRS. Furthermore, if we are to add a new equation e to the rewrite system $\mathcal{R}$ during the exploration of the theory then we must prove termination of the extended rewrite system $\mathcal{R} \cup \{e\}$. To this end, we use the termination technique described in section 3.3. Note however, that a terminating rewrite system could produce different normal forms for the same term, depending on the order in which rules are applied. Even the position in a term

[1] Note that '+' denotes standard addition of naturals.

to which a rule is applied could affect its normal form. For example, the terminating rewrite system $\mathcal{R} := \{f\ (f\ x) \to g\ x\}$ induces two different normal forms on the term $f\ (f\ (f\ x))$ depending on the position where the rule is applied.



This is in fact a critical pair which is automatically detected and oriented as $f\ (g\ x) \to g\ (f\ x)$ by completion (with the recursive path order induced by the precedence $f > g$). Completion yields the convergent rewrite system $\{f\ (f\ x) \to g\ x,\ f\ (g\ x) \to g\ (f\ x)\}$. Critical pairs are usually helpful, particularly by bringing to light interesting lemmas which may not be apparent at first glance: e.g., $f\ (g\ x) = g\ (f\ x)$ is a useful and not obvious consequence of $f\ (f\ x) = g\ x$. This example also demonstrates that a convergent rewrite system is more useful than a terminating one, if we are to reduce redundancies in the search for conjectures and definitions. The following definition will help with the description of the algorithm for theory-exploration found in section 4.5.

Definition 8. *Given a terminating rewrite system $\mathcal{R}$ and an instantiation $i \in Insts(s,X)$ of the form $\forall\overline{x}.\ s = t$, the* **normalising extension** *$ext(\mathcal{R},i)$ of $\mathcal{R}$ with i is denoted by*

$$ext(\mathcal{R},i) := \begin{cases} \mathcal{R}' & \text{if completion succeeds for } \mathcal{R} \cup \{s=t\} \\ & \text{with a convergent system } \mathcal{R}' \\ \mathcal{R} \cup \{r\} & \text{otherwise if termination succeeds for } \mathcal{R} \cup \{r\} \\ & \text{with } r \in \{s=t,\ t=s\} \\ \mathcal{R} & \text{otherwise.} \end{cases}$$

Example 9. *Given $\mathcal{R} := \{x*(y+z) = (x*y)+(x*z)\}$ and $i := \forall u\,v\,w.\,(u+v)*w = (u*w)+(v*w)$, then $R \cup \{(u+v)*w = (u*w)+(v*w)\}$ cannot be shown to be convergent because completion produces the non-orientable identity $(u*y)+(v*y)+(u*z)+(v*z) = (u*y)+(u*z)+(v*y)+(v*z)$ (see section 3.4). However, termination succeeds with the recursive path order induced by the precedence $+ > *$ producing the normalising extension $ext(\mathcal{R},i) := \{x*(y+z) \to (x*y)+(x*z), (u+v)*w \to (u*w)+(v*w)\}$.*

Theorem 2. *The normalising extension $ext(\mathcal{R},i)$ of a terminating rewrite system $\mathcal{R}$ and an instantiation i of the form $\forall\overline{x}.\,s = t$ is terminating.*

Proof. There are exactly three cases in the definition of the normalising extension $ext(\mathcal{R}, i)$ of a terminating rewrite system $\mathcal{R}$ and an instantiation i of the form $\forall\overline{x}.\, s = t$. The first case is when completion succeeds and $ext(\mathcal{R}, i)$ is convergent. Since there is an ordering $>$ such that the rule Orientation in table 3.2 orients all rules, termination is guaranteed. The second is when we find an ordering $>$ such that $R \cup r$ with $r \in \{s = t, t = s\}$. This case obviously implies termination. The third case denotes the normalising extension as $\mathcal{R}$ and termination is proved by assumption. □

4.4 Filtering of Conjectures and Definitions

As suggested by definition 8, IsaScheme updates the rewrite system $\mathcal{R}$ each time a new equational theorem is found. It is thus useful to consider the notion of equivalence of instantiations modulo $\mathcal{R}$.

Definition 9. *Let u and v be two instantiations and $\mathcal{R}$ a terminating rewrite system.* **Equivalence of instantiations** *modulo $\mathcal{R}$ is denoted as*

$$u \approx_R v := (\hat{u} \approx_\alpha \hat{v})$$

where $\hat{u}$ and $\hat{v}$ are normal forms (w.r.t. $\mathcal{R}$) of u and v respectively and $\approx_\alpha$ is term equivalence up to variable renaming.

The problem of identifying equivalent instantiations is just an instance of the so-called *word problem*[2]. This word problem is in general undecidable. Definition 9 tries to solve instances of it through normalisation. However, even if $\mathcal{R}$ is convergent, two equivalent terms can have distinct normal forms. Consider, for example, the following defining equations for addition of natural numbers

$$\begin{aligned} 0 + y &= y \\ suc(x) + y &= suc(x + y). \end{aligned} \tag{4.8}$$

Clearly the equations (4.8) form a convergent rewrite system but the latter would fail to decide equivalence of the terms $x + suc(y)$ and $suc(x + y)$ which are already in normal form w.r.t. (4.8). The problem is that rewriting is not sufficient for inductive theories; here proofs by induction are often required [10].

The aforementioned problem leads to redundancies in the construction of definitions (see example 10). It is thus useful to consider the notion of equivalence of definitions.

[2] Given a set of identities $\mathcal{R}$ and two terms s and t, is it possible to transform the term s into the term t using the equations in $\mathcal{R}$ in both directions?

Definition 10. *Let $f, g \in \mathcal{F}$ be two function symbols with type $\tau_1 \to \ldots \to \tau_n \to \tau_0$. We say that f and g are* **equivalent definitions** *iff*

$$\forall \overline{x}.\ f\ \overline{x} = g\ \overline{x}.$$

Example 10. *Given $X := \{(\lambda x.\, x), suc\}$ and s_d the following definitional scheme*

$$\left(\begin{array}{l} def\text{-}scheme\text{-}nat\ G\ H\ I \equiv \\ \qquad \exists f.\ \forall xy.\quad \bigwedge \left\{ \begin{array}{l} f\ 0\ y = G\ y \\ f\ (suc\ x)\ y = H\ (f\ x\ (I\ y)) \end{array} \right. \end{array} \right),$$

the schematic substitutions $\sigma_1 = \{G \mapsto (\lambda x.\, x), H \mapsto suc, I \mapsto (\lambda x.\, x)\}$ and $\sigma_2 = \{G \mapsto (\lambda x.\, x), H \mapsto (\lambda x.\, x), I \mapsto suc\}$ produce the instantiations depicted in table 4.3.

$\sigma \in Sub(s_d, X)$	$inst(s_d, \sigma_i)$
$\sigma_1 = \left\{ \begin{array}{c} G \mapsto (\lambda x.\, x), \\ H \mapsto suc, I \mapsto (\lambda x.\, x) \end{array} \right\}$	$\exists f.\ \forall xy.\ \bigwedge \left\{ \begin{array}{l} f\ 0\ y = y \\ f\ (suc\ x)\ y = suc\ (f\ x\ y) \end{array} \right.$
$\sigma_2 = \left\{ \begin{array}{c} G \mapsto (\lambda x.\, x), \\ H \mapsto (\lambda x.\, x), I \mapsto suc \end{array} \right\}$	$\exists f.\ \forall xy.\ \bigwedge \left\{ \begin{array}{l} f\ 0\ y = y \\ f\ (suc\ x)\ y = f\ x\ (suc\ y) \end{array} \right.$

Table 4.3: Two different instantiations syntactically which induce equivalent definitions.

If we assume that $inst(s_d, \sigma_1)$ and $inst(s_d, \sigma_2)$ are already in normal form with respect to some $\mathcal{R}$, then $inst(s_d, \sigma_1) \not\approx_{\mathcal{R}} inst(s_d, \sigma_2)$. Moreover, the defining equations of the instantiations $inst(s_d, \sigma_1)$ and $inst(s_d, \sigma_2)$ induce equivalent definitions (addition of natural numbers in this case) as shown in table 4.4:

$\sigma \in inst(s_d, \sigma_i)$	Induced defining equations
$inst(s_d, \sigma_1)$	$f_1\ 0\ y = y$ $f_1\ (suc\ x)\ y = suc\ (f_1\ x\ y)$
$inst(s_d, \sigma_2)$	$f_2\ 0\ y = y$ $f_2\ (suc\ x)\ y = f_2\ x\ (suc\ y)$

Table 4.4: Two different ways of defining addition of natural numbers.

4.4.1 Argument Neglecting Definitions

Since the exploration process could generate a substantial number of definitions and each of them could potentially produce a multitude of conjectures it becomes necessary to restrict the search space in some way. For example, instead of generating $f_1\ x\ y = x^2$ and $f_2\ x\ y = y^2$ it would be better to just generate $f\ x = x^2$ and construct f_1 and f_2 on top of f, e.g. $(\lambda xy.\ f\ x)$ and $(\lambda xy.\ f\ y)$. Preliminary studies suggested that a significant proportion of functions synthesised by schemes would ignore one or more of their arguments. Such argument neglecting functions can always be defined with another function using fewer arguments and a λ-abstraction. This motivates the following definition, which we use in section 4.5, to avoid generating such definitions.

Definition 11. *Let $f \in \mathcal{F}$ be a function with type $\tau_1 \to \ldots \to \tau_n \to \tau_0$, where τ_0 is a base type and $n > 0$. Then f is an* **argument neglecting function** *provided that*

$$\forall \overline{x_n} yz.\ f\ x_1\ \ldots\ x_{k-1}\ y\ x_{k+1}\ \ldots\ x_n = f\ x_1\ \ldots\ x_{k-1}\ z\ x_{k+1}\ \ldots\ x_n$$

for some k where $1 \leq k \leq n$.

Note that in definition 11, τ_0 is required to be a base type. The following example will help illustrate the need for this restriction.

Example 11. *Let $f : nat \to nat \to nat$ be a function defined as:*

$$\begin{aligned} f\ 0\ y &= 0 \\ f\ (suc\ x)\ y &= suc\ x. \end{aligned}$$

Then f is an argument neglecting function because $\forall x_1 yz.\ f\ x_1\ y = f\ x_1\ z$, which can be proved by induction on x_1. Referring to definition 11, clearly $n = 2$ and $\tau_0 = nat$ is a base type.

Observe that f (in example 11) could be viewed as $f : nat \to \tau_0$ where $\tau_0 = nat \to nat$ – were it not for the requirement that τ_0 be of base type. Note that in this view, f does not neglect its (only) argument.

4.5 Theory-Exploration Algorithms

4.5.1 Introduction

We have developed a program for synthesising conjectures and definitions, which we call IsaScheme. IsaScheme automatically generates instantiations from a set of

schemes and a set of closed terms (in a theory $\mathcal{T}$). Every instantiation is then normalised w.r.t. a terminating rewrite system $\mathcal{R}$ constructed from the defining equations of definitions in $\mathcal{T}$ and the lemmata discovered during theory-exploration. False conjectures are filtered out with counter-example checking. Given that the number of instantiations is exponential as shown by theorem 1, counter-example checking can be a rather slow process. However, we found that different substitutions can lead to the same instantiation (module $\mathcal{R}$). This suggested the use of definition 9 to test if a new instantiation was already processed, thus avoiding counter-example checking on equivalent instantiations. The test would involve simply rewriting, which is faster than counter-example checking. Unfalsified conjectures are given to a parametric prover to prove the conjecture. Instantiations from definitional schemes are given to Isabelle's function package to test for well-definedness. New theorems and defining equations are then used to extend the rewrite system $\mathcal{R}$, thus improving the normalisation process.

In section 4.5.2, we describe the algorithm for the generation of theorems used by IsaScheme. This algorithm is in turn used recursively by the algorithm which generates definitions in IsaScheme, described in section 4.5.3.

4.5.2 Scheme-based Conjecture Synthesis.

The overall procedure for the generation of theorems is described by the pseudocode of `InventTheorems` (see figure 4.3). The algorithm receives as arguments a proof method (or tactic) $\mathcal{P}$, a set of terms I (conjectures), a terminating rewrite system $\mathcal{R}$, a set of terms T (theorems), a set of schemes S_p and a set of closed terms X_p with which schemes are to be instantiated. Note that initially, $I = T = \varnothing$.

The algorithm iterates through all instantiations obtained from schemes in S_p and the terms X_p. Each instantiation is normalised w.r.t. $\mathcal{R}$ in line 2. Line 3 ensures that the instantiation is neither subsumed by any previously proved theorem nor trivially proved by simplification. Equivalent instantiations modulo $\mathcal{R}$ are identified in line 4. This line can be implemented efficiently using discrimination nets (or by a constraint mechanism as in [33]) and avoids counterexample checking equivalent instantiations modulo $\mathcal{R}$. Falsifiable instantiations are detected in line 5 to avoid any proof attempt on such conjectures. Isabelle/HOL provides the counter-example checkers *Quickcheck* [14] and *Nitpick* [4] which are used to refute the false conjectures in the implementation of *IsaScheme*. Although the tools are not complete, they succeed in

```
   InventTheorems(𝒫, I, ℛ, T, S_p, X_p)
1  for each i ∈ ⋃_{s ∈ S_p} Insts(s, X_p)
2    î := a normal form of i w.r.t. ℛ
3    if î is not subsumed by T ∪ {True} and
4    there is not a j ∈ I such that j ≈_R î and
5    cannot find a counter-example of î then
6      if 𝒫 can prove î then
7        T := T ∪ {î}
8        if î is of the form ∀x̄. s = t then ℛ := ext(ℛ, î)
9    I := I ∪ {î}
10 return <I, ℛ, T>
```

Figure 4.3: Algorithm describing the generation process of theorems.

spotting most falsifiable conjectures in practice. In case the conjecture is not rejected by the inspection in lines 3, 4 or 5 then a proof attempt is performed in line 6. The prover used for the proof obligations is the parameter $\mathcal{P}$ of the algorithm. We evaluated the algorithm in *IsaScheme* (see chapter 7) with the automatic proof techniques described in chapter 5. Following a successful proof attempt, line 7 adds the theorem into the set of theorems T. Line 8 will update $\mathcal{R}$ with its normalising extension if the theorem proved is an equation. The set of processed instantiations is updated in line 9.

In the implementation of IsaScheme, the `InventTheorems` algorithm also used the rules depicted in table 4.5 to perform normalisation of instantiations. These rules are configured as simplification rules in Isabelle/HOL by default to improve proof automation.

The algorithm `InventTheorems` is necessarily not complete as it may not terminate in general because of lines 6 and 8[3] (an opportunity for diverging techniques such as [65]). For example, completion in line 8 does not always terminate as it could produce infinitely many critical pairs during its execution (e.g. the HRS $f\ (g\ (f\ x)) \rightarrow f\ (g\ x)$ generates infinitely many critical pairs of the form $f\ (g^n\ (f\ x)) \rightarrow f\ (g^n\ x)$ during completion). In fact, IsaScheme uses timeouts in those critical lines to avoid non-termination.

[3]Note that line 2 always terminates as a consequence of theorem 2.

$x = (x = f) \equiv f$	$\forall x.\ t = x \longrightarrow P\ x \equiv P\ t$	$\forall x.\ x = t \longrightarrow P\ x \equiv P\ t$
$\exists x.\ t = x \wedge P\ x \equiv P\ t$	$\exists x.\ x = t \wedge P\ x \equiv P\ t$	$\exists x.\ t = x \equiv True$
$\exists x.\ x = t \equiv True$	$\exists x.\ y \equiv y$	$\forall x.\ y \equiv y$
$P \vee P \vee Q \equiv P \vee Q$	$y \vee y \equiv y$	$False \vee y \equiv y$
$y \vee False \equiv y$	$True \vee P \equiv True$	$P \vee True \equiv True$
$\neg P \wedge P \equiv False$	$P \wedge \neg P \equiv False$	$P \wedge P \wedge Q \equiv P \wedge Q$
$y \wedge y \equiv y$	$False \wedge P \equiv False$	$P \wedge False \equiv False$
$True \wedge y \equiv y$	$y \wedge True \equiv y$	$P \longrightarrow \neg P \equiv \neg P$
$P \longrightarrow False \equiv \neg P$	$P \longrightarrow P \equiv True$	$P \longrightarrow True \equiv True$
$False \longrightarrow P \equiv True$	$True \longrightarrow y \equiv y$	$False = P \equiv \neg P$
$True = y \equiv y$	$P = (\neg P) \equiv False$	$(\neg P) = P \equiv False$
$\neg False \equiv True$	$\neg True \equiv False$	$x = x \equiv True$
$\neg P \vee P \equiv True$	$P \vee \neg P \equiv True$	$P\neg = Q \equiv P = (\neg Q)$
$(\neg P) = (\neg Q) \equiv P = Q$	$\neg\neg y \equiv y$	$(P = False) \equiv (\neg P)$
$(P = True) \equiv P$		

Table 4.5: Rewrite rules used to perform normalisation in IsaScheme.

4.5.2.1 Irreducibility of $\mathcal{R}$

Collecting specialised versions of theorems can be unwieldy and does not add anything new to the discovery process. As a consequence of lines 2 and 3 of the `InventTheorems` algorithm, whenever we prove a theorem t, t is a $\mathcal{R}$-normal form not subsumed by a previously proved theorem. However, t may be a generalisation of a previously proved one. Completion in definition 8 handles this situation returning always an *irreducible*[4] rewrite system. However, Knuth-Bendix completion does not always succeed and termination checking itself cannot discard specialised versions of proved theorems. The algorithm depicted in figure 4.4 can be used instead of simple termination checking in definition 8. It will try to transform a non-irreducible rewrite system into an irreducible one. The second case of definition 8 can now be lifted to obtain a potentially irreducible rewrite system $\mathcal{R}'$ if $Irreducible(NONE, \mathcal{R} \cup \{r\})$ succeeds with $SOME\ \ R'$[5] where $r \in \{s = t, t = s\}$ (see definition 12). As expected, the algorithm `Irreducible` is not complete. However, it often succeeds in practice.

[4] A HRS $\mathcal{R}$ is **irreducible** if for any rule $l \to r \in R$, l and r are normal forms in $\mathcal{R} \setminus \{l \to r\}$.

[5] An option type is used for handling partial functions and optional values. Its definition is $datatype\ 'a\ option\ =\ NONE \mid SOME\ of\ 'a$.

```
   Irreducible(ℛ′, ℛ)
1  if ℛ is terminating then
2     ℛ″ := {l̂ → r̂ | ∃ l → r ∈ ℛ. l̂ is some ℛ \ {l → r}-normal form of l,
                                  r̂ is some ℛ \ {l → r}-normal form of r,
                                  and l̂ ≠ r̂}
3     if ℛ ≡ ℛ″ then return SOME ℛ
4     else Irreducible(SOME ℛ, ℛ″)
5  else return ℛ′
```

Figure 4.4: Algorithm to transform a non-irreducible rewrite system into an irreducible one.

The algorithm `Irreducible` receives as arguments an *option type* argument $\mathcal{R}'$ corresponding to an optional terminating rewrite system in the first argument, and a potentially non-terminating rewrite system $\mathcal{R}$ in the second argument. Initially, the first argument is *NONE*. The algorithm first tests for termination of $\mathcal{R}$ in line 1. If $\mathcal{R}$ is not proved to be terminating, the algorithm returns the first argument $\mathcal{R}'$ in line 5. If $\mathcal{R}$ is shown to be terminating, the algorithm normalises the left and right sides of each equation in $\mathcal{R}$ w.r.t. the rest of the rules (excluding the rule itself being normalised). Trivial rules are eliminated in the process and the new potentially non-terminating rewrite system is stored in $\mathcal{R}''$. If all rules in $\mathcal{R}$ are still in $\mathcal{R}''$, then $\mathcal{R}$ is already irreducible and terminating and the algorithm returns *SOME* $\mathcal{R}$ in line 3. Otherwise, the rewrite system $\mathcal{R}''$ may be non-terminating, and an additional termination check will be carried out in the recursive call $Irreducible(SOME\ \mathcal{R}, \mathcal{R}'')$ of the algorithm in line 4.

Definition 12. *Given a terminating rewrite system $\mathcal{R}$ and an instantiation $i \in Insts(s, X)$ of the form $\forall \overline{x}.\, s = t$, the* **normalising extension** *$ext(\mathcal{R}, i)$ of $\mathcal{R}$ with i is denoted by*

$$ext(\mathcal{R}, i) := \begin{cases} \mathcal{R}' & \text{if completion succeeds for } \mathcal{R} \cup \{s = t\} \\ & \text{with a convergent system } \mathcal{R}' \\ \mathcal{R}' & \text{otherwise if } Irreducible(NONE, \mathcal{R} \cup \{r\}) \text{ succeeds} \\ & \text{with some } R' \text{ where } r \in \{s = t,\ t = s\} \\ \mathcal{R} & \text{otherwise.} \end{cases}$$

Definition 12 is useful in eliminating specialised versions of theorems in $\mathcal{R}$. IsaScheme also performs a similar sanity check on theorems not in $\mathcal{R}$ and which are stored in T by the `InventTheorems` algorithm.

4.5.2.2 Unfalsified and Unproved Conjectures

Theorems about inductively defined data structures and recursive definitions usually require induction to prove them. Inductive proving is, in general, undecidable and the failure of cut elimination for inductive theories implies that new lemmas or generalisations are often needed [10]. The requirement of new lemmas or generalisations suggests that whenever we prove a new theorem we increase the chances of succeeding in a previously failed proof attempt.

IsaScheme keeps a cache of unfalsified and unproved conjectures that are revisited every time a new proof is found. For the sake of clarity, this is not explicitly described in figure 4.3.

4.5.2.3 AC-rewriting

Rules such as commutativity of addition and multiplication are inherently non-terminating. Definitions 8 or 12 fail to exploit such rules because the ordering described in section 3.3 cannot orientate those equations. However, it is known that such rules can be handled in the context of *ordered rewriting* [43, 60]. The rewrite relation under ordered rewriting needs a reduction order $\succ$ as in termination. Termination is enforced by admitting a rewrite step only if it decreases the term w.r.t. $\succ$. Identities can be used in both directions because termination is enforced in each rewrite step (provided the order decreases). The requirement for $\succ$ is that it needs to be total on ground terms. This does not pose any problem rewriting terms with variables because we can always replace all variables by new free constants [1].

In [43] the authors proved that the following ordered rewrite system is ground convergent (provided $>$ is total on ground terms).

$$f\ (f\ x\ y)\ z \rightarrow f\ x\ (f\ y\ z) \tag{4.9}$$

$$f\ x\ y = f\ y\ x \qquad \text{if } x > y \tag{4.10}$$

$$f\ x\ (f\ y\ z) = f\ y\ (f\ x\ z) \qquad \text{if } x > y \tag{4.11}$$

In fact, Isabelle's Simplifier uses a predefined ordering on ground terms during rewriting when it is supplied with permutative rules (see section 2.3.5) such as commutativity. IsaScheme exploits this by identifying the rules (4.9), (4.10) and (4.11) for some symbol f during theory-exploration. Once these rules are all found, IsaScheme fixes the status of (4.10) and (4.11) as ordered rewrite rules with the Simplifier. Note that rule (4.10) could prevent the formation of rules (4.9) and (4.11) if used on them.

For instance, the term $f\ (f\ a\ b)\ c = f\ a\ (f\ b\ c)$ would have been simplified to $f\ c\ (f\ a\ b) = f\ a\ (f\ b\ c)$ by (4.10). For this reason IsaScheme waits until the three rules are discovered to configure (4.10) and (4.11) as permutative rules with the Simplifier.

An important aspect of the design of IsaScheme is that it does not attempt to construct any theorem eagerly. The reason for this is that we did not want to hard-code the construction of any theorem inside IsaScheme and avoid hiding things from the user. Consequently, in order for IsaScheme to use ordered rewriting effectively with AC-operators, the schemes used during the exploration of the theory must be able to generate the above rules. For example, we could use the following schemes with the `InventTheorems` algorithm to produce the aforementioned rules.

$$\begin{aligned}
&\textit{assoc-scheme}\ P \equiv \forall xyz.\ P\ (P\ x\ y)\ z = P\ x\ (P\ y\ z)\\
&\textit{comm-scheme}\ P \equiv\ \forall xy.\ P\ x\ y = P\ y\ x\\
&\textit{lcomm-scheme}\ P \equiv\ \forall xyz.\ P\ x\ (P\ y\ z) = P\ y\ (P\ x\ z)
\end{aligned}$$

Alternatively, we could use a more general scheme subsuming the previous ones, such as s_2 in example 12 of section 4.5.2.4.

4.5.2.4 Examples of the InventTheorems Algorithm

Below we describe an execution trace of the algorithm `InventTheorems` (and the additions of sections 4.5.2.1, 4.5.2.2 and 4.5.2.3) with a theory of naturals with addition. As a consequence of the high number of synthesised conjectures (6244 conjectures in this case), we only report unfalsified conjectures.

Example 12. *Let $\mathcal{F}$ be a signature consisting of $\mathcal{F} := \{suc : nat \to nat, 0 : nat, + : nat \to nat \to nat\}$. To keep things simple we assume an omniscient proof procedure $\mathcal{P}$ which finds a proof if it exists (although we know from the incompleteness result of Gödel that this is not possible in general). Also let s_1 and s_2 be the following schemes (here we assume the most general types inferred for the schemes).*

$$s_1 := \left(\begin{array}{l}\textit{assoc-scheme}\ P \equiv\\ \qquad \forall xyz.\ P\ (P\ x\ y)\ z = P\ x\ (P\ y\ z)\end{array}\right)$$

$$s_2 := \left(\begin{array}{l}\textit{scheme-binary}\ P\ Q\ R\ S\ T\ U \equiv\\ \qquad \forall xyz.\ P\ (Q\ x\ y)\ (R\ x\ z) = S\ (T\ x\ z)\ (U\ y\ z)\end{array}\right)$$

and assuming the following set of closed terms X

$$X := \left\{\begin{array}{c}(\lambda xy.\ x), (\lambda xy.\ y), (\lambda x.\ suc), (\lambda xy.\ suc\ x),\\ (\lambda xy.\ 0), (\lambda xy.\ suc\ 0), +\end{array}\right\}.$$

Moreover, let $\mathcal{R}$ be the convergent rewrite system

$$\mathcal{R} := \left\{ \begin{array}{rcl} 0+y & \to & y \\ (suc\ x)+y & \to & suc\ (x+y) \end{array} \right\}.$$

The first unfalsified conjecture is obtained with the substitution $\sigma_1 := \{P \mapsto +\}$ on s_1. σ_1 produces the associativity theorem for addition with the instantiation $inst(s_1, \sigma_1) := \forall xyz.\,(x+y)+z = x+(y+z)$ which is proved by $\mathcal{P}$. The algorithm `InventTheorems` *now constructs the normalising extension of $\mathcal{R}$ and $\forall xyz.\,(x+y)+z = x+(y+z)$ producing a new $\mathcal{R}$ after a successful execution of completion*

$$\mathcal{R} := \left\{ \begin{array}{rcl} 0+y & \to & y \\ (suc\ x)+y & \to & suc\ (x+y) \\ (x+y)+z & \to & x+(y+z) \end{array} \right\}.$$

The second unfalsified conjecture is obtained with the substitution $\sigma_2 := \{P \mapsto +, Q \mapsto (\lambda xy.\,x), R \mapsto (\lambda xy.\,0), S \mapsto (\lambda xy.\,x), T \mapsto (\lambda xy.\,x)\}$ on s_2. This substitution produces the commuted version of the base case of addition with the instantiation $inst(s_2, \sigma_2) := \forall x.\,x+0 = x$ which is proved by $\mathcal{P}$. The normalising extension of $\mathcal{R}$ and $\forall x.\,x+0 = x$ produces a new $\mathcal{R}$ after a successful execution of completion

$$\mathcal{R} := \left\{ \begin{array}{rcl} 0+y & \to & y \\ (suc\ x)+y & \to & suc\ (x+y) \\ (x+y)+z & \to & x+(y+z) \\ x+0 & \to & x \end{array} \right\}.$$

The third substitution $\sigma_3 := \{P \mapsto +, Q \mapsto (\lambda xy.\,x), R \mapsto (\lambda xy.\,suc\ 0), S \mapsto (\lambda xy.\,x), T \mapsto (\lambda xy.\,suc\ x)\}$ on s_2 produces the instantiation $inst(s_2, \sigma_3) := \forall x.\,x+(suc\ 0) = suc\ x$ which is proved by $\mathcal{P}$. The normalising extension of $\mathcal{R}$ and $\forall x.\,x+(suc\ 0) = suc\ x$ produces a new $\mathcal{R}$ after a successful execution of completion

$$\mathcal{R} := \left\{ \begin{array}{rcl} 0+y & \to & y \\ (suc\ x)+y & \to & suc\ (x+y) \\ (x+y)+z & \to & x+(y+z) \\ x+0 & \to & x \\ x+(suc\ y) & \to & suc\ (x+y) \end{array} \right\}.$$

Note that the theorem $x+(suc\ 0) = suc\ x$ is discarded by completion and in turn produces a more general version of the theorem automatically, i.e. the commuted version of the step case of addition $x+(suc\ y) = suc\ (x+y)$.

The fourth substitution $\sigma_4 := \{P \mapsto +, Q \mapsto (\lambda xy.\ y), R \mapsto (\lambda xy.\ x), S \mapsto +, T \mapsto (\lambda xy.\ x), U \mapsto (\lambda xy.\ x)\}$ *on* s_2 *produces the instantiation* $inst(s_2, \sigma_4) := \forall xy.\, x+y = y+x$ *(again proved by* $\mathcal{P}$*). This instantiation invents commutativity of addition which is required by AC-rewriting. However, this equation cannot be oriented by the techniques described in section 3.3 and also cannot be used as an ordered rewriting rule because IsaScheme has not found the equivalent equation (4.11) for addition also required by AC-rewriting. Thus, completion and termination fail leaving* $\mathcal{R}$ *unchanged.*
The fifth substitution $\sigma_5 := \{P \mapsto +, Q \mapsto +, R \mapsto (\lambda xy.\ y), S \mapsto +, T \mapsto +, U \mapsto (\lambda xy.\ x)\}$ *on* s_2 *produces the instantiation* $inst(s_2, \sigma_5) := \forall xyz.\, x+(y+z) = x+(z+y)$*. Here the subterm* $z+y$ *is not simplified to* $y+z$ *because commutativity of addition, previously discovered, is not configured as an ordered rewrite rule yet because IsaScheme has not found the equivalent equation (4.11) for addition (see section 4.5.2.3). This instantiation is another permutative rule such as commutativity and cannot be oriented by the techniques described in section 3.3. Hence, completion and termination fail again leaving* $\mathcal{R}$ *unchanged.*
The sixth substitution $\sigma_6 := \{P \mapsto +, Q \mapsto (\lambda xy.\ y), R \mapsto +, S \mapsto +, T \mapsto +, U \mapsto (\lambda xy.\ x)\}$ *on* s_2 *produces the instantiation* $inst(s_2, \sigma_6) := \forall xyz.\, x+(z+y) = y+(x+z)$*. This instantiation again is a permutative consequence of associativity and commutativity of addition and cannot be oriented by the techniques described in section 3.3. Hence, completion and termination fail again leaving* $\mathcal{R}$ *unchanged.*
The seventh substitution $\sigma_7 := \{P \mapsto +, Q \mapsto (\lambda xy.\ y), R \mapsto +, S \mapsto +, T \mapsto (\lambda xy.\ x), U \mapsto +\}$ *on* s_2 *produces the instantiation* $inst(s_2, \sigma_7) := \forall xyz.\, x+(y+z) = y+(x+z)$*. This is the last equation required by AC-rewriting and thus IsaScheme turns the relevant equations into ordered rewrite rules producing a new* $\mathcal{R}$

$$\mathcal{R} := \left\{ \begin{array}{rcl} 0+y & \rightarrow & y \\ (suc\ x)+y & \rightarrow & suc\ (x+y) \\ (x+y)+z & \rightarrow & x+(y+z) \\ x+0 & \rightarrow & x \\ x+(suc\ y) & \rightarrow & suc\ (x+y) \\ x+y & \leftrightarrow & y+x \\ x+(y+z) & \leftrightarrow & y+(x+z) \end{array} \right\}.$$

Note that the theorems produced by the instantiations σ_5 *and* σ_6 *are now normalised to* $True$ *(w.r.t.* $\mathcal{R}$ *and the rules in table 4.5) and thus are discarded by IsaScheme.*
The `InventTheorems` *algorithm finishes its execution producing a total of 6244*

instantiations. There are 287 falsified conjectures and a total of 5702 equivalent instantiations (see section 4.4). Only 248 instantiations were subsumed by the theorems synthesised. The set T of new theorems discovered by `InventTheorems` *is:*

$$T := \left\{ \begin{array}{rcl} (x+y)+z & = & x+(y+z) \\ x+0 & = & x \\ x+(suc\ y) & = & suc\ (x+y) \\ x+y & = & y+x \\ x+(y+z) & = & y+(x+z) \end{array} \right\}.$$

4.5.3 Scheme-based Definition Synthesis.

The generation of definitions is described by the pseudocode of the algorithm `InventDefinitions` (see figure 4.5). The algorithm takes as input the same arguments received by the `InventTheorems` method. Additionally, it also takes a set of function symbols $\mathcal{F}$ in the current theory, a set of terms D (definitions), a set of definitional schemes S_d and a set of closed terms X_d from which definitional schemes are to be instantiated. Again, initially $D = \varnothing$.

The algorithm iterates through all instantiations obtained from definitional schemes in S_d and terms in X_d. Each instantiation d is reduced to a normal form $\hat{d}$ w.r.t. $\mathcal{R}$ and the rules in table 4.5 in line 2. Since $\hat{d}$ is generated from a definitional scheme, it has the form $\exists f_1 \ldots f_n . \forall \overline{y} . e_1 \wedge \ldots e_m$ where $f_1, \ldots, f_n$ are variables standing for the new functions to be defined and $e_1, \ldots, e_m$ are the defining equations of the functions. In lines 4 and 5, new function symbols $f'_1, \ldots, f'_n$ (w.r.t. the signature $\mathcal{F}$) are created and a substitution σ is constructed to uniquely denote each of the new functions to be defined. This 'renaming' of functions is performed with the defining equations and $[e'_1, \ldots, e'_m]$ is obtained in line 6. Line 7 ensures that definitions which are equivalent modulo $\mathcal{R}$ to earlier generated ones, are ignored. Well-definedness properties, such as existence and uniqueness of the functions generated, are proved in line 8. We used Isabelle/HOL's *function package* [39] for these proof obligations. Line 9 checks if the new functions created are not argument neglecting (AN). Definitions that are identified to be equivalent, by theorem proving, to previously defined functions are rejected in line 10. Prior to any proof attempt of the conjectures demanded by lines 9 and 10 (definitions 10 and 11), we try to produce counter-examples of those functions being equivalent or AN. In case an instantiation $\hat{d}$ is not rejected by lines 7, 8, 9 or 10, then the context $\mathcal{F}$ and the theorems T are updated with the new function symbols $f'_1, \ldots, f'_n$ and the theorems $\{e'_1, \ldots, e'_m\}$ respectively (lines 11 and 12). Line 13 updates the

```
InventDefinitions(𝒫, I, ℛ, T, S_p, X_p, ℱ, D, S_d, X_d)
1  for each d ∈ ⋃_{s ∈ S_d} Insts(s, X_d)
2     d̂ := a normal form of d w.r.t. ℛ
3     let ∃f_1 ... f_n.∀ȳ. e_1∧...e_m = d̂
4     create new function symbols f'_1,...,f'_n such that f'_i ∉ ℱ
5     σ := {f_1 ↦ f'_1,...,f_n ↦ f'_n}
6     [e'_1,...,e'_m] := [σ(e_1),...,σ(e_m)]
7     if there is not a j ∈D such that j ≈_R d̂ then
8     if [e'_1,...,e'_m] is well-defined then
9     if f'_1,...,f'_n are not argument neglecting then
10    if f'_1,...,f'_n are not equivalent to previous functions then
11       ℱ := ℱ ∪ {f'_1,...,f'_n}
12       T := T ∪ {e'_1,...,e'_m}
13       ℛ := ℛ ∪ {e'_1,...,e'_m}
14       <I, ℛ, T>:=InventTheorems(I, ℛ, T, S_p, X_p ∪ {f'_1,...,f'_n})
15    D := D ∪ {d̂}
16 return <I, ℛ, T, ℱ, D>
```

Figure 4.5: Algorithm describing the generation process of definitions.

rewrite system $\mathcal{R}$ with the newly introduced defining equations $e'_1,\ldots,e'_m$. A call to `InventTheorems` is performed in line 14 updating $I,\mathcal{R}$ and T. At the end of each iteration, the instantiation $\hat{d}$ is added to the set of processed definitions D in line 15. Finally, when all instantiations $d \in \bigcup_{s\in S_d} Insts(s,X_d)$ have been processed, the values $<I,\mathcal{R},T,\mathcal{F},D>$ are returned. To illustrate this algorithm in action, we present a session using a theory of naturals in appendix B.

4.5.3.1 Delaying the Rejection of Definitions

In general, it is difficult and expensive to prove the conjectures demanded by definitions 10 and 11, especially if we do not know anything about the new function to be analysed. However, analogously to section 4.5.2.2, we improve the chances of succeeding in these proof obligations after the exploration of line 14 of the `InventDefinitions` algorithm.

In the implementation of IsaScheme, we delayed tackling the proof obligations of lines 9 and 10, demanded by definitions 10 and 11 respectively, in case of a previously

failed proof attempt.

4.6 Automatic generation of closed terms

In section 4.2, we described a technique to instantiate schemes automatically given a set of closed terms. Notice that while the user has to provide these closed terms, they are formulaic in nature. They consist of projections and constructor symbols (possibly surrounded by λ-abstractions). For instance, example 12 used the set of closed terms

$$X := \left\{ \begin{array}{c} (\lambda xy.\ x), (\lambda xy.\ y), (\lambda x.\ suc), (\lambda xy.\ suc\ x), \\ (\lambda xy.\ 0), (\lambda xy.\ suc\ 0), + \end{array} \right\}$$

to instantiate the schemes

$$s_1 := \left(\begin{array}{l} \mathit{assoc\text{-}scheme}(P) \equiv \\ \quad \forall xyz.\ P\ (P\ x\ y)\ z = P\ x\ (P\ y\ z) \end{array} \right)$$

$$s_2 := \left(\begin{array}{l} \mathit{scheme\text{-}binary}\ P\ Q\ R\ S\ T\ U \equiv \\ \quad \forall xyz.\ P\ (Q\ x\ y)\ (R\ x\ z) = S\ (T\ x\ z)\ (U\ y\ z) \end{array} \right).$$

Projection functions can be inferred from the arity of free variables in schemes. For instance, the free variable $P : \tau \to \tau \to \tau$ of scheme s_1 is a binary operator with arity 2. This arity can be used to infer some obvious projection functions to be used during the instantiation process of the scheme s_1 such as $(\lambda xy.\ x)$ and $(\lambda xy.\ y)$. Here, ternary projection functions such as $(\lambda xyz.\ x)$, are not an option because these are not unifiable with P.

Arities in the free variables of schemes can be used not only to infer 'common' projection functions, but also to 'accommodate' function symbols (or other closed terms which in the sequel we will denote as *terms of interest*) with incompatible arities w.r.t. those in the free variables of schemes. For example, the function symbol $suc : nat \to nat$ is unary and thus, is not unifiable with $P : \tau \to \tau \to \tau$. However, we can wrap *suc* around a λ-abstraction to construct a closed term unifiable with P. For instance, the closed terms $\lambda xy.\ suc\ x$ and $\lambda xy.\ suc\ y$ (which is η-equivalent to the term $\lambda x.\ suc$) have types $nat \to \tau \to nat$ and $\tau \to nat \to nat$ respectively and thus, are unifiable with $P : \tau \to \tau \to \tau$.

The overall procedure for the automatic generation of closed terms is described by the pseudocode of `ClosedTerms` (see figure 4.6). The algorithm receives as arguments a scheme s and a set of terms of interest X with which the scheme is to be instantiated.

The algorithm uses the auxiliary function *BinomialSelection* which is informally defined as follows. Given a set s and a number k, $BinomialSelection(k,s)$ is a function returning the set of k-combinations or arrangements of the elements in s. For example, $BinomialSelection(2, \{x,y,z\}) := \{(x,y), (x,z), (y,x), (y,z), (z,x), (z,y)\}$.

```
   ClosedTerms(s,X)
1  C := ∅
2  for each v ∈ 𝒱(s)
3    k := arity of free variable v
4    C := C ∪ projection functions with arity k
5    for each t ∈ X
6      l := arity of t
7      if l ≤ k then
8        x̄ := k new fresh variables
9        for each ȳ ∈ BinomialSelection(l, x̄)
10         C := C ∪ {(λx̄. t ȳ)}
11 return C
```

Figure 4.6: Algorithm describing the generation process of closed terms from a set of 'terms of interest' provided by the user.

Line 1 of the *ClosedTerms* algorithm performs the initialization of C which will contain the generated closed terms at the end of the execution. The algorithm then iterates through all free variables of the scheme s. In line 3, the arity of each free variable is stored in the local variable k. This arity is then used to construct all projection functions with arity k in line 4. For instance, if $k = 3$ then the projection functions generated are $(\lambda xyz.\ x)$, $(\lambda xyz.\ y)$ and $(\lambda xyz.\ z)$. The inner loop of line 5 iterates through each term of interest $t \in X$ and its arity is stored in l (line 6). In case $l \leq k$ a list $\overline{x}$ of k new fresh variables (w.r.t. t) is created in lines 7 and 8. For each l-combination $\overline{y}$ of the variables in $\overline{x}$, the new closed term $(\lambda\overline{x}.\ t\ \overline{y})$ is added to the set C (lines 9 and 10). Finally, the generated closed terms C are returned in line 11. For example, if the algorithm *ClosedTerms* is called with the scheme s_2 and the terms of interest $X := \{0, suc, suc\ 0, +\}$, then the output of the algorithm is the set of closed terms

$$X := \left\{ \begin{array}{c} (\lambda xy.\ x), (\lambda xy.\ y), (\lambda x.\ suc), (\lambda xy.\ suc\ x), \\ (\lambda xy.\ 0), (\lambda xy.\ suc\ 0), +, (\lambda xy.\ y+x) \end{array} \right\}.$$

The terms of interest are thus a point of interaction with the user in IsaScheme.

The user can now, for example, select those function symbols that he(she) wishes to use during the exploration process and free him(her) from constructing complicated lambda terms.

4.7 Applicability and Higher-order Capabilities

An important aspect of every theory-exploration system is its applicability across different mathematical theories. The scheme-based approach provides a generic mechanism for the exploration of any mathematical theory where the symbols in the theory's signature and the free variables within the schemes could be unified. As an illustration, consider the definitional scheme

$$\left(\begin{array}{l} \mathit{def}\text{-}\mathit{scheme2}\ G\ H\ I\ J\ K \equiv \\ \qquad \exists f.\ \forall x y z.\quad \bigwedge \left\{ \begin{array}{l} f\ G\ y = H\ y \\ f\ (I\ z\ x)\ y = J\ (K\ x\ y\ z)\ (f\ x\ y) \end{array} \right. \end{array} \right)$$

which generates all previous definitions from example 14 (assuming the same set of closed terms). Also let $\mathcal{T}_{\mathcal{L}}$ be the theory of lists with the constructors $[\,]$ and # (the empty list and cons, respectively) and the definition of function composition $\circ$ ($f \circ g = \lambda x.\, f\ (g\ x)$). The definition of the higher-order function *map*[6] can be generated by using the substitution $\{G \mapsto [\,], H \mapsto (\lambda x.\ [\,]),\ I \mapsto \#, J \mapsto \#, K \mapsto (\lambda x y.\ y)\}$ on the aforementioned definitional scheme, producing

$$\exists f.\ \forall x y z. \bigwedge \left\{ \begin{array}{l} f\ [\,]\ y = [\,] \\ f\ (z\ \#\ x)\ y = (y\ z)\ \#\ (f\ x\ y). \end{array} \right.$$

Applicability and higher-order capabilities are not confined to definitional schemes. Schemes can also be used to create higher-order conjectures in different mathematical theories. The substitution $\{P \mapsto \mathit{map}, Q \mapsto \mathit{map}, R \mapsto (\lambda x y.\ y), S \mapsto \mathit{map}, T \mapsto (\lambda x y.\ x), U \mapsto (\lambda x y.\ y \circ x)\}$ on the scheme

$$\left(\begin{array}{l} \mathit{scheme}\text{-}\mathit{binary}\ P\ Q\ R\ S\ T\ U \equiv \\ \qquad \forall x y z.\ P\ (Q\ x\ y)\ (R\ x\ z) = S\ (T\ x\ z)\ (U\ y\ z) \end{array} \right)$$

generates a conjecture $\mathit{map}\ z\ (x \circ y) = \mathit{map}\ (\mathit{map}\ z\ y)\ x$ about *map* and $\circ$ which states that the *map* of the function composition of x and y over the list z is equal to the *map* of the function x over the list obtained from the *map* of y over the list z.

[6] The function *map* applies a given function to all elements of a list and returns the list of results. This function is higher-order because in the second equation, y appears both as a variable and as a function symbol.

4.8 Completion and Proof by Mathematical Induction

Proofs by mathematical induction are hard because they often require to speculate unknown lemmas. The discovery of unknown lemmas is a challenging problem for inductive theorem proving and have generally been assumed to require user intervention. The automated discovery of inductive lemmas has been extensively used in verifying properties of both software and hardware. Because of its importance, methods to generate lemmas on demand have been implemented in different inductive theorem provers such as SPIKE [65], CLAM [30], IsaPlanner [34], etc.

Completion is a sound method which has proved to be very successful for solving word problems in equational theories [1]. It has also been used for inductive theorem proving in the context of inductionless induction (also called proofs by consistency) [62]. The algorithm `InventTheorems` from section 4.5.2 combines theory-exploration techniques with completion to maintain (if possible) a convergent set of rules which can be reused for proving other goals. In the case of inductive theories, the algorithm has proved to be a very successful approach to find useful lemmas (see section 3.5 and chapter 7) using a simple inductive tactic (see section 5.4 of chapter 5).

4.9 Summary

Theorem discovery and concept invention are vital for theory-exploration systems. This chapter presented a formal description of the generation of theorems and definitions grounded in the concept of a scheme. We described an algorithm to automatically instantiate a scheme with a set of closed terms in a theory. In the worst case complexity of this algorithm, the number of instantiations was found to be exponential, but we mediated this problem using a terminating rewrite system $\mathcal{R}$ to work only on instantiations in $\mathcal{R}$-normal form. The rewrite system $\mathcal{R}$ was constructed from the defining equations of definitions and the lemmata discovered during the exploration of the theory, using the termination and completion techniques described in sections 3.3 and 3.4 respectively.

We implemented two algorithms performing theory-exploration in IsaScheme. The `InventTheorems` algorithm was used to synthesise conjectures (in $\mathcal{R}$-normal form), which were then given to a counterexample checker to avoid trivially false statements. Proofs were attempted on the unfalsified conjectures, and the equational lemmata found were used to extend the normalisation machinery. The `InventDefinitions` al-

gorithm was used to synthesise (definitional) instantiations in $\mathcal{R}$-normal form, which were then given to the function package to ensure well-definedness of new definitions. Argument neglecting functions or functions shown to be equivalent (by theorem proving) to previously defined functions were rejected. The remaining functions were then used for further theory-exploration in a recursive call to the `InventTheorems` algorithm.

Chapter 5

Inductive Proof Automation

Theorems about inductively defined data structures and recursive definitions usually require induction. Inductive proving is in general undecidable and the failure of cut elimination for inductive theories implies that new lemmas or generalizations are often needed [10]. In this chapter we describe the proof technique we used during the exploration of theories with IsaScheme. Our main motivation to construct this tactic was to exploit and test the techniques described in chapter 4 without having to bring in the whole proof planning infrastructure. We compare this proof method with the inductive prover IsaPlanner which uses the rippling heuristic [11]. In the first part we overview rippling for inductive proof search. In the second part we describe our simple inductive proof technique for Isabelle that has shown to be very useful when combined with the term rewrite system $\mathcal{R}$ described in section 4.3.

5.1 Rippling

Rippling is a proof technique, originally developed for proving by mathematical induction, that works by the selective application of rewrite rules. Rippling manipulates one formula, the *goal*, to make it resemble another, *the given* [11]. At this point, the given can be used to help prove the goal in a method called *fertilization*.

Rippling removes the differences between the goal and the given using annotations on the goal called *wave annotation*. For instance, suppose our given and goal are (example taken from [23])

$$\begin{array}{rrcl} \text{Given:} & \forall b.\ a+b & = & b+a \\ \text{Goal:} & (suc\ a)+b & = & suc\ (b+a) \end{array}$$

and that we want to use the given to prove the goal. One possible annotation of the goal is:

$$\boxed{(suc\ a)}^{\uparrow} + \lfloor b \rfloor \quad = \quad \boxed{suc\ (\lfloor b \rfloor + a)}^{\downarrow}.$$

The constituent parts of this annotated term are described as follows. The *skeleton* of an annotated goal is the part of the formula to be preserved. It is written with no annotation and must be a well-formed formula. The *wave-fronts*, which are not well-formed formulas, are those parts of the goal to be moved. They are annotated in a grey box with an arrow at the top right, which indicates the required direction of the wave-front. A *wave-hole* denotes a sub-term inside a wave-front that is part of the skeleton. *Sinks* indicate positions in the skeleton that correspond to universally quantified variables and are indicated as grey boxes. The directions a wave-front can pursue are inward and outward, and they are called *inward wave-fronts* (rippling-in) and *outward wave-fronts* (rippling-out) respectively. Rippling-in tries to move a difference into a sink and rippling-out tries to move towards the top of the term tree.

Rippling is provided with a measure that ensures its termination. This measure is defined as a well-founded order on annotated terms. Termination is enforced by admitting a rewrite step only if the measure decreases. This allows rippling to use identities in both directions because termination is enforced in each rewrite step. There are several measures in the literature [23] but we will concentrate on a measure based on the sum of distances (sum-of-distances) from outward wave-fronts to the top of the term tree and from inward wave-fronts to the nearest sink. In figure 5.1 this measure is exemplified with the annotated term $\boxed{(suc\ a)}^{\uparrow} + \lfloor b \rfloor = \boxed{suc\ (\lfloor b \rfloor + a)}^{\downarrow}$, with sum-of-distances measure 4.

In the situation where different rewritings apply for the same goal, the heuristic is to choose the rewriting with smaller measure. To illustrate rippling in action, we take a simple example from [11] p. 9. Consider the following set of rewrite rules:

$$(X + Y) + Z \rightarrow X + (Y + Z) \tag{5.1}$$

$$(X_1 + X_2 = Y_1 + Y_2) \rightarrow (X_1 = Y_1 \wedge X_2 = Y_2), \tag{5.2}$$

and suppose our given and goal formulas are:

Given: $a + b = 42$

Goal: $(\ \boxed{(c+d)+a}^{\uparrow}\ ) + b = \boxed{(c+d)+42}^{\uparrow}$ sum-of-distances=3

Thus we can rewrite our goal, for instance, with the rule (5.1) in three different ways:

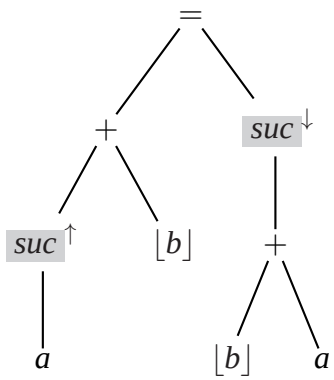


Figure 5.1: Annotated term tree where the given is $\forall b.\, a+b = b+a$ and the goal is $(suc\ a)+b = suc\ (b+a)$. The sum-of-distances measure of the term is 4. The distance from the outward wave-front to the top of the term tree is 2 and the distance from the inward wave-front the nearest sink is 2.

$$\begin{aligned}(c+d)+(a+b) &= (c+d)+42\\ ((c+d)+a)+b &= c+(d+42)\\ (c+(d+a))+b &= (c+d)+42\end{aligned}$$

If we annotate each of the rewritings with respect to the given, $a+b=42$, we get:

$$(c+d)+(a+b)^{\uparrow} = (c+d)+42^{\uparrow} \quad \text{sum-of-distances=2} \tag{5.3}$$

$$(\,(c+d)+a)^{\uparrow}\,)+b = c+(d+42)^{\uparrow} \quad \text{sum-of-distances=3} \tag{5.4}$$

$$(\,c+(d+a)^{\uparrow}\,)+b = (c+d)+42^{\uparrow} \quad \text{sum-of-distances=3} \tag{5.5}$$

and now we can decide which rewriting has the smaller measure – namely the rule application (5.3). In fact, the rule applications (5.4) and (5.5) are examples of unwanted rewritings because they do not decrease the sum-of-distance measure of the annotated goal (which is 3). We can now apply rewrite rule (5.2) to goal (5.3) to obtain the new annotated goal

$$c+d=c+d \wedge a+b=42^{\uparrow} \quad \text{sum-of-distances=1} \tag{5.6}$$

and this clearly shows progress with respect to the wave measure which is now 1. The goal (5.6) contains now an instance of the given and we can proceed with *fertilization*. Since the given is supposed to be true, we can replace its instance in (5.6) to T. This gives $c+d=c+d \wedge T$, which is trivial to prove.

5.1.1 IsaPlanner

IsaPlanner is a proof-planner [9] for Isabelle which uses the rippling heuristic. It was developed originally by Lucas Dixon as part of his PhD [23], and recently extended with proof planning critics [30] by Moa Johansson [32]. IsaPlanner combines proof-planning with the execution of the proof in Isabelle following the LCF-style tradition, and thus, resulting in a conservative extension of Isabelle. IsaPlanner's proof states (*reasoning states*) contain information of the proof strategy (plan) constructed so far, the next tactic to apply, and other contextual information such as annotated rules, discovered lemmas, etc.

5.2 Induction Schemes in Isabelle

Isabelle provides induction schemes for ML-style inductive datatypes, namely *structural induction* rules. Inductive datatypes are implemented by the datatype package [3]. Common types such as naturals, lists or trees can be defined as inductive datatypes in Isabelle. For example, the naturals can be defined with the command:

datatype `nat = zero ("0") | suc nat`

Isabelle will automatically derive the appropriate induction scheme for the datatype. For example, the theorem representing the induction scheme for this datatype is:

$[\![$`?P 0; ⋀n. ?P n ⟹ ?P (suc n)`$]\!] \Longrightarrow$ `?P ?n`

Recall that variables prefixed by a question mark are meta-variables that are allowed to be instantiated by unification, $\bigwedge$ is universal quantification, $\Longrightarrow$ stands for meta-implication and the term $[\![P; Q]\!] \Longrightarrow R$ abbreviates $P \Longrightarrow (Q \Longrightarrow R)$ (see section 2.3).

Isabelle also provides induction schemes following the recursive structure of functions, namely *recursion induction* rules. This is implemented by the function package [39] (see section 2.3.6). The definition of recursive functions is similar to other definitions in Isabelle. For example, addition for natural numbers can be defined with the following command:

```
fun add :: "nat ⇒ nat ⇒ nat" (infix "+" 70) where
"0 + y = y" |
"(suc x) + y = suc (x + y)"
```

The function package will automatically derive an induction scheme for this definition which is:

```
⟦⋀y. ?P 0 y; ⋀x y. ?P x y ⟹ ?P (suc x) y⟧ ⟹ ?P ?a0.0 ?a1.0
```

The `induct_tac` tactic can be used to apply these induction schemes in Isabelle. The parameters of this tactic comprise a list of subterms in the goal on which induction is to be performed, together with an induction scheme. Although `induct_tac` performs induction on a subterm of the goal, provided the induction scheme is appropriate for the subterm's type, it is commonly used only on variables of the goal. This tactic tries to unify the conclusion of the induction rule with the goal. If successful, it yields new subgoals given by the antecedents of the rule.For instance, if the induction scheme for the naturals is applied to x on the goal $x+y=y+x$, the induction tactic leaves the subgoals $0+y=y+0$ and $\bigwedge n.\ n+y=y+n \Longrightarrow suc\ n+y=y+suc\ n$ (instantiating $P \mapsto \lambda a.\ a+y=y+a$).

The `induct_tac` is of great help for interactive theorem proving because of the freedom it provides to the user, but it can be problematic when used in an automated proof method. The reason is that unification can produce more than one unifier of the goal with the conclusion of the induction rule, hence producing a combinatorial explosion. For example, unification in the above application of `induct_tac` does not produce a unique unifier ($P \mapsto \lambda a.\ a+y=y+a$). In fact, it produces different ones such as, for example, $P \mapsto \lambda a.\ x+y=y+a$ which produces the subgoals $x+y=y+0$ and $\bigwedge n.\ x+y=y+n \Longrightarrow x+y=y+suc\ n$. The former subgoal is obviously false and a proof cannot be found.

5.3 Case-statements in Isabelle

Isabelle supports case expressions on user-defined datatypes. For each datatype, the datatype package creates a case-construct, which can be used for pattern matching, and proves some properties about it [3]. For instance, the case constructor for natural numbers is called $nat\_case : \sigma \to (nat \to \sigma) \to nat \to \sigma$. The case-expression

$$case\ x\ of\ 0 \Rightarrow f_1 \mid suc\ y \Rightarrow f_2\ y$$

is internally represented in Isabelle by the term $nat\_case\ f_1\ f_2\ x$. The properties automatically proved about nat_case are, among others,

$$(case\ 0\ of\ 0 \Rightarrow f1 \mid suc\ x \Rightarrow f2\ x) = f1 \tag{5.7}$$

$$(case\ suc\ n\ of\ 0 \Rightarrow f1 \mid suc\ x \Rightarrow f2\ x) = f2\ n. \tag{5.8}$$

For a datatype `t`, these rules can be accessed with the name `t.cases` in Isabelle. Another automatically derived rule produced by the datatype package is the *split-rule* of the datatype (accessed with the name `t.split`). For the naturals this rule is:

$$\begin{array}{c} ?P\ (nat\_case\ ?f1\ ?f2\ ?x) = \\ ((?x = 0 \longrightarrow ?P\ ?f1) \wedge (\forall nat.\ ?x = suc\ nat \longrightarrow ?P\ (?f2\ nat))) \end{array} \tag{5.9}$$

Split-rules are usually used if neither of the 'case' rules (such as (5.7) and (5.8)) can be applied to the goal. For instance, the goal $(case\ ?x\ of\ 0 \Rightarrow 0 \mid suc\ n \Rightarrow suc\ n) = ?x$ cannot be reduced with (5.7) or (5.8). However, if we 'case-split' the goal with (5.9), we produce the new goal $(?x = 0 \longrightarrow 0 = ?x) \wedge (\forall n.\ ?x = suc\ n \longrightarrow suc\ n = ?x)$ which can be solved by simplification. The `split_tac` tactic can be used to apply split rules in Isabelle. The parameter of this tactic is the split-rule of the datatype.

5.4 The Induction and Simplification Tactic

The top-level tactic essentially performs induction, case splitting and then simplification in a best-first search. Best-first search is implemented by Isabelle's tactical `BEST_FIRST` which we guide by a heuristic given by the number of pending goals.

We now describe the induction tactic used by the top-level tactic. The `Induct_auto_tac` receives as arguments a terminating rewrite system $\mathcal{R}$ and a goal g. Pseudocode for the tactic is presented below.

```
   Induct_auto_tac(R, g)
1  tac := no_tac
2  g' := Conclusion of g
3  for each v ∈ V(g') ∪ B(g')
4    if v is of an inductively defined type then
5      structural_rule := Structural induction rule for v
6      tac := tac APPEND induct_tac([v], structural_rule)
7  for each f v̄ ∈ { t | there exists p ∈ Pos(g') such that g'|_p = t }
8    if v̄ are all variables and f is defined by recursion then
9      recursive_rule := Recursive induction rule for f
10     tac := tac APPEND induct_tac(v̄, recursive_rule)
11   else if f is a case constructor then
12     split_rule := Split rule for f
13     tac := tac APPEND split_tac(split_rule)
```

14 **return** $(TRY\ \ tac)\ \ THEN\ \ (TRY\ \ auto\_tac(\mathcal{R}))$

The induction tactic first collects all variables in the conclusion of the goal that are of an inductively defined type and performs structural induction on them (lines 3-6). The tactic then traverses all subterms $f\ \overline{v}$ in the conclusion of the goal in line 7. It performs recursive induction on $\overline{v}$ using the recursion induction rule for f if f is defined by recursion and $\overline{v}$ is a list of variables (line 10). If f is a case constructor then case-splitting is applied to the goal (line 13). After induction (and/or case-splitting) is completed, the tactic calls Isabelle's `auto_tac` with $\mathcal{R}$ as its argument, performing rewriting on all pending subgoals.

5.5 Evaluation

This sections presents an evaluation of the tactic described in section 5.4. We evaluate the tactic and compare its performance with IsaPlanner on a set of 92 example theorems taken from [32]. Most of the theorems were in turn taken from a subset of inductive theorems from Isabelle's libraries for lists and naturals, some were taken from a corpus for the CLAM system [30] and from problems emerged from dependently typed programming [67].

For the experiments, we conducted two sets of tests: one where both inductive tactics were only supplied with the definitions given in Appendix A (the minimal lemma configuration), and one with the proved theorems available to the tactics (the maximal lemma configuration). Since the `Induct_auto_tac` performs rewriting exhaustively, it will become ineffective when a non-terminating rewrite system is given to the tactic. Fortunately, we already have the machinery to take care of this problem. We used the `InventTheorems` algorithm to orientate the proved equational theorems. For this, we performed theory-exploration using the scheme $scheme(C) \equiv C$ along with the set of 92 conjectures as the closed terms (we explicitly quantify free variables in the conjectures).

We compare both inductive techniques w.r.t. *success rate* (which technique more often leads to a proof) and *timing* (which technique is faster in CPU time), in a setting similar to [12]. The experiments were run in a GNU/Linux node with 2 dual core CPUs and 4GB of RAM memory. We also used Isabelle/2009-2 and IsaPlanner svn version 3031 for the experiments. Each proof attempt had a timeout of 180 seconds.

5.5.1 Results

The overall statistics are given in table 5.1. The success rates of both inductive provers were roughly the same in both lemma configurations. In the minimal lemma configuration, the rippling tactic proved 48 theorems and the `Induct_auto_tac` proved 52, giving success rates of 52.1% and 56.5% respectively. The number of theorems proved by both inductive provers increased in the maximal lemma configuration. Rippling proved 65 theorems and the `induct_auto_tac` tactic proved 64, giving success rates of 70.7% and 69.6% respectively. We knew that the `Induct_auto_tac` prover would fail when bi-directional rewriting was required. Here rippling might have better chances to succeed. However, we were surprised to see that the numbers of theorems proved in both configurations were almost the same. In the minimal lemma configuration, a possible reason for this is that the opportunities for bi-directional rewriting are rare when the rules are restricted to the defining equations of definitions [12]. The increment of the success rates in the maximal lemma configuration for both provers is not surprising since the opportunities to succeed in a previously failed proof attempt grow as more lemmas are available. In the maximal lemma configuration we not only use the defining equations of definitions and the proved theorems for rewriting, we also use the critical pairs discovered during completion. For instance, two additional properties of if were added after completion at some point: $(if\ True\ then\ h \# t\ else\ [\,]) = h \# t$ and $(if\ False\ then\ t\ else\ h \# t) = h \# t$.

Statistic	Rippling		Reduction	
	Min	Max	Min	Max
Overall success (%)	52.1	70.7	56.5	69.6
Average CPU time (seg)	0.330	1.1	1.001	5.85

Table 5.1: Overall statistics for rippling and the `Induct_auto_tac`.

We expected rippling to be faster than the simplification-based prover because of the overhead of using a special recursion induction rule for each function symbol in the goal. While different induction rules are supported by IsaPlanner, we only used structural induction rules. Although we did not measure branching factors for both provers, we believe that the rippling heuristic produces a smaller branching factor than the heuristic given by the numbers of pending goals used by the `Induct_auto_tac` tactic. On average, the `Induct_auto_tac` tactic was 3 times slower than rippling in the

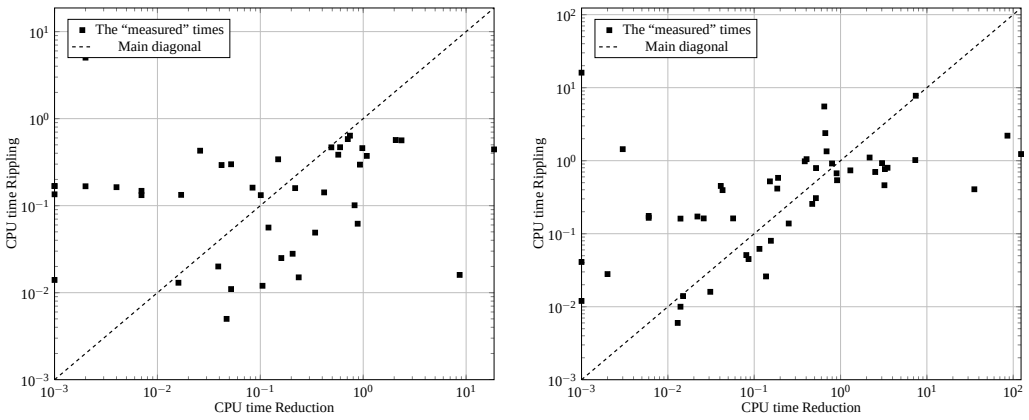


Figure 5.2: A point (x,y) in the graphs illustrates the CPU time spent during proofs using the simplification-based prover (component x) and IsaPlanner (component y). The main diagonal (dashed line) indicates the situation where both provers spent the same amount of time for each proof. The scattergram at the left refers to the minimal lemma configuration and the one at the right to the maximal lemma configuration.

minimal lemma configuration and 5 times slower in the maximal lemma configuration. The scattergrams in figure 5.2 show a graphical view of the time spent during proofs by both tactics for the minimal lemma configuration (left) and the maximal lemma configuration (right). The evaluation corpus and results are included in Appendix C.

There are two main differences between the two tactics. Firstly, IsaPlanner contains critics that can generate lemmas on demand when rippling is blocked or when a generalization is needed. An example where this property is needed is with the theorem $height\ (mirror\ n) = height\ n$ (where n is a tree). By induction on n and then rewriting, rippling gets blocked with the subgoal:

$$suc\ (max\ (height\ x)\ (height\ y)) = suc\ (max\ (height\ y)\ (height\ x)).$$

At this point, IsaPlanner's critics get triggered producing the generalization $suc\ (max\ x\ y) = suc\ (max\ y\ x)$ which IsaPlanner proves and uses to finish the proof. This useful technique is however, not implemented in the `induct_auto_tac` tactic and thus, it fails to prove the theorem $height\ (mirror\ n) = height\ n$ as commutativity of max is required. Note that even in the maximal lemma configuration where all theorems are available for the tactics (including commutativity of max which is proved by both tactics), the `induct_auto_tac` tactic fails to prove the theorem. The reason is that the rule is not exploited by IsaScheme because it is non-terminating and thus is not provided to the Simplifier. In fact, IsaScheme waits until it finds the complete set of

rules required for AC-rewriting (see section 4.5.2.3) but this never happens because the required rule $max\ x\ (max\ y\ z) = max\ y\ (max\ x\ z)$ is never discovered. By contrast, section 7.4 describes a theory-exploration on trees, during which the theorem $height\ (mirror\ n) = height\ n$ is proved by the `induct_auto_tac` tactic because the set of rules required for AC-rewriting is found by IsaScheme.

Secondly, if the skeleton does not embed one of the new sub-goals then rippling gets blocked and IsaPlanner tries to solve the sub-goal by using new lemmas or generalizations. However, this emergency technique does not always succeed even when the sub-goal can be solved by simplification. Here, the `induct_auto_tac` might have better chances to succeed. A case where this happens is with the theorem $((suc\ m) - n) - (suc\ k) = (m - n) - k$. For instance, after induction on m, IsaPlanner is left at some point with the sub-goal:

$$(case\ n\ of\ 0 \Rightarrow suc\ 0 \mid suc\ y \Rightarrow 0 - y) - (suc\ m) = 0 \tag{5.10}$$

Performing induction on n in sub-goal (5.10) gives the following step-case:

$$\begin{aligned} IH: &\quad (case\ n\ of\ 0 \Rightarrow suc\ 0 \mid suc\ y \Rightarrow 0 - y) - (suc\ m) = 0 \\ Goal: &\quad (case\ \boxed{suc\ n}^{\uparrow}\ of\ 0 \Rightarrow suc\ 0 \mid suc\ y \Rightarrow 0 - y) - (suc\ m) = 0 \end{aligned}$$

The goal can be solved by simplification using the following rules (case rule (5.8) and the base-case of the definition of *minus*):

$$\begin{aligned} (case\ suc\ n\ of\ 0 \Rightarrow f1 \mid suc\ x \Rightarrow f2\ x) &= f2\ n \\ 0 - y &= 0. \end{aligned}$$

Defining equations and datatype cases are used as simplification rules by default in Isabelle's Simplifier. The problem is that these rules are not measure-decreasing as the induction hypothesis has a case-statement in it. This prevents rippling from using the rules. An easy solution could be to perform simplification on step-cases when rippling and proof planning critics fail. Since simplification is assumed to be terminating in IsaPlanner, this extended technique will retain termination.

5.6 Summary

In this chapter we described a simple inductive proof technique for Isabelle that has shown to be very useful when combined with the term rewrite system described in Chapter 4. We compared this technique with the inductive prover IsaPlanner on a set

of 92 example theorems. Our evaluation showed that the success rate of our inductive proof technique rivals that of IsaPlanner. However, IsaPlanner was shown to be faster than our proof technique. A possible reason for this is the overhead of using a special recursion induction rule for each function symbol in the goal increasing the branching factor of the search.

Chapter 6

IsaScheme Architecture and Design

We have presented a scheme-based technique to generate conjectures and definitions in a mathematical theory in sections 4.1 and 4.2. We also performed a selection of conjectures and definitions based on normalisation techniques in section 4.3 and 4.4. The generation and identification techniques were then described algorithmically in section 4.5. In this chapter we present details regarding the design and implementation of our approach, split into three sections:

1. Details regarding the synthesis of conjectures and definitions (instantiations).

2. Details on the normalisation of instantiations and the associated completion and termination techniques.

3. Details on the identification module and its associated tools.

IsaScheme's source code and all the results described in chapter 7 are provided at `http://sourceforge.net/projects/isaplanner/` and `http://dream.inf.ed.ac.uk/projects/isascheme/` respectively.

IsaScheme is implemented in Standard ML (SML) where most of the functions used are supplied by Isabelle's ML function library. These functions are broadly composed by unification and matching for terms and types, type inference, parsing facilities for terms, types and tactics (inner syntax), parsing facilities for new user defined Isar commands (outter syntax), access to definitional packages such as the function package and access to counter-example checkers such as Quickcheck, Nitpick and Refute. Currently, IsaScheme totals over 10000 lines of SML code and ensures accuracy in repeated empirical tests by exploring different theories and inspecting the results.

The architecture and design of IsaScheme closely follows the presentation of the theory detailed in chapter 4 and is illustrated in figure 6.1. Control flows between three core modules that implement the synthesis of conjectures and definitions (Synthesis module), normalisation of instantiations and the associated completion and termination algorithms (Normalisation module), and the filtering of conjectures and definitions using subsumption and counter-example checking (Identification module).

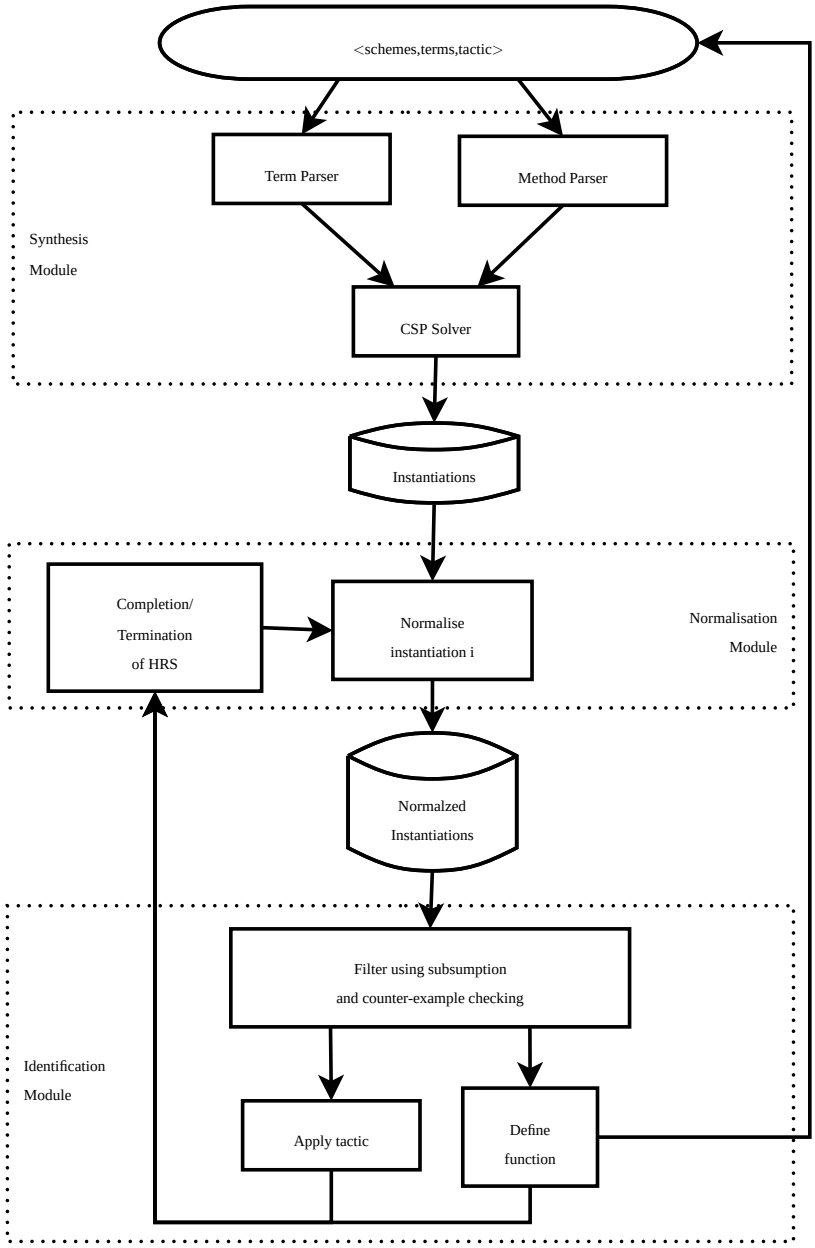


Figure 6.1: The flow of control between modularized components in IsaScheme.

6.1 Synthesis of Conjectures and Definitions

Our implementation of the synthesis of conjectures and definitions is mainly composed by a constraint satisfaction solver (CSP Solver) and two parsing modules responsible for the parsing of terms and the parsing of isabelle tactics (called methods). Isabelle distinguishes between *outer* and *inner* syntax. Terms, types and so on belong to the inner syntax, whereas commands, such as `fun`, `datatype`, `definition` and so on belong to the outter syntax. Isabelle developers are usually concerned with writing outer syntax parsers, either for new user commands or for calling methods with specific arguments. IsaScheme defines two new Isabelle commands for the execution of the algorithms described in section 4.5: `InventTheorems` and `InventDefinitions`. A typical theory file using IsaScheme is depicted in figure 6.2. In this theory, the `InventTheorem` command receives as input the scheme S, the set of closed terms $\{\lambda x\ y.\ x,\ \lambda x\ y.\ y,\ add\}$, the method `induct_auto` (see section 5.4) and the optional theorem attributes `simp` to add a terminating rewrite system to the Simplifier and `simp del` to delete rules from the Simplifier that potentially loops or because they are no longer needed as a result of a new more general rewrite rule set.

6.1.1 Constraint Satisfaction Solver

The CSP Solver takes the parsed closed terms and schemes as input, and perform the search for schematic substitutions (see definition 5 of section 4.2). The solver begins with an initialization of the domains of the free variables in the scheme(s). The domains represent the closed terms that can be unified to each of the variables. An iterative process then follows. Variables are instantiated sequentially and if a partial instantiation leaves no possible values for a variable then backtracking is performed to the most recently instantiated variable that still has alternatives available. Each time a variable is instantiated, the remaining domains are updated w.r.t. the most general unifier of the last instantiation reducing the search space.

A key component for the efficient implementation of the CSP Solver is the ability to identify variables that are removed or eliminated by previous instantiations after β-contraction. For example, the term $\forall x\ y.\ P\ (Q\ x\ y)\ (R\ x\ y)$ has three free variables: P, Q and R. If the term is instantiated with the substitution $\{P \mapsto (\lambda x\ y.\ x)\}$ and then β-contraction is applied, the term $\forall x\ y.\ Q\ x\ y$ is obtained where the variable R is no longer required to be instantiated. To exploit the identification of removed variables, IsaScheme imposes an ordering in which variables are instantiated. The order consists

```
theory Addition
imports IsaScheme
begin

datatype N = zero ("0") | suc N

definition scheme ("S") where
"S P Q ≡ (∀ x y z. Q x (P y z) = P (Q x y) (Q x z))"

fun add :: "N ⇒ N ⇒ N" (infix "+" 70) where
"(0::N) + y = y" |
"(suc x) + y = suc (x + y)"

InventTheorems[simp]:[simp del]:
prop_schemes "S"
prop_terms "λx y. x" | "λx y. y" | "add"
method induct_auto
end
```

Figure 6.2: Typical theory file using the `InventTheorems` command of IsaScheme.

on the length of the position (see section 3.2) of each variable, instantiating variables at upper levels in the term tree first.

6.2 Normalisation Module

The normalisation module consists of three main components. A component responsible for the normalisation of instantiations, a component that performs the completion of a rewrite system and a component that performs termination checking.

6.2.1 Normalisation of Instantiations

The normalisation of instantiations component takes as input an instantiation generated by the Constraint Satisfaction Solver. This instantiation is then normalised w.r.t. a rewrite system generated by the components that perform completion and termination. The normalisation process consists on exhaustive rewriting using Isabelle's Simplifier. Ordered rewriting, required by AC-rewriting (see section 4.5.2.3), is performed by

simplification procedures. Simplification procedures, also called *simprocs*, implement custom simplification procedures that are triggered by the Simplifier on a specific term-pattern and rewrite a term according to a theorem. IsaScheme uses *simprocs* to handle AC-rules as they potentially loops.

6.2.2 Completion of a Rewrite System

The component performing the completion of a rewrite system takes as input a set of (meta) equations and a new theorem/equation to be added to the rewrite system. All function symbols $c_1, c_2, \ldots, c_n$ are then extracted from the equations and a precedence on them $c'_1 < c'_2 < \ldots < c'_n$ is found such that the induced recursive path ordering is compatible with the equations.

Instead of enumerating all possible precedences of n function symbols, which produce $n!$ possible precedences, we encode the problem into a set of linear inequalities in HOL and then we call the counter-example checker Refute to determine the satisfiability of the formula. Refute then translates the formula into propositional logic and calls different SAT Solvers which usually solve the satisfiability problem in under a second. Empirical results show the efficiency of the approach as opposed to enumerating the possible precedences.

The inference rules for completion from section 3.4 are then applied to the set of equations using the recursive path ordering found by the SAT Solvers and if a fixed point (E, R) is reached where $E = \varnothing$ then a convergent rewrite system R is produced. The rewrite system R is then stored globally so that other modules and components of the system can access the convergent rewrite system.

6.2.3 Termination of a Rewrite System

The completion algorithm, described in 3.4, does not always succeed on a set of identities and a reduction ordering. In those cases IsaScheme performs a termination check to ensure that the normalisation process terminates. The termination component takes as input a set of (meta) equations and a new theorem/equation to be added to the rewrite system. As in completion, IsaScheme produce a compatible recursive path ordering with the set of equations (including the new equation) and if the ordering is found then the new extended rewrite system is terminating and can be used to find normal forms.

6.3 Identification Module

Our implementation of the Identification Module is mainly composed by three components: filtering of conjectures and definitions, the application of tactics, and the definition of new recursive functions.

6.3.1 Subsumption and Counter-example Checking

Two different schematic substitutions σ_1 and σ_2 can produce equivalent instantiations (see section 4.3). For this reason IsaScheme stores the normal forms of the instantiations produced by the CSP Solver. Whenever a new normalised instantiation is produced, IsaScheme checks if the instantiation was not previously generated and filters it accordingly. IsaScheme uses existing Isabelle tools such as discrimination nets and matching to provide a fast and economic subsumption of instantiations. IsaScheme also use discrimination nets to check if a new instantiation is an instance of an already proved theorem. All instances of theorems are discarded by IsaScheme.

Counter-example checking is then applied at non-filtered conjectures. For this we use Quickcheck and Nitpick as they usually provide a good range of applicability for conjectures involving recursive functions and datatypes.

6.3.2 Application of Methods

Methods are central to Isabelle. They are used whenever an **apply** command is written in an Isabelle theory file. The available Isabelle methods are displayed using the **print_methods** command and all of them can be used in IsaScheme. IsaScheme uses existing Isabelle parsers to parse proof methods and they are then used to discharge proof obligations. These proof obligations include the conjectures not filtered by subsumption and counter-example checking, the detection of argument neglecting functions and the detection of equivalent definitions (see definitions 10 and 11 of section 4.4) during the exploration of the theory. Unfalsified and unproven conjectures are kept in cache and they are revisited every time a new proof is found (see section 4.5.2.2).

6.3.3 Definition of Recursive Functions

As described in section 4.1, definitional schemes are used to generate new recursive definitions. However, definitional schemes could generate non-terminating functions

(see section 2.2). For this reason, IsaScheme uses Isabelle's function package to safely define general recursive function definitions.

Definitional instantiations not filtered by subsumption are sent to the function package to test for well-definedness. Since the exploration process could generate a substantial number of definitions and each of them could potentially produce a multitude of conjectures, IsaScheme filters out recursive functions that ignore one or more of their arguments, so call *argument neglecting functions*. IsaScheme also checks whether the new definition is equivalent to a previously defined function and discards it accordingly. The defining equations of not filtered well-defined functions are then added to the global rewrite system to exploit them during normalisation.

6.4 Summary

IsaScheme is a SML-based implementation of theory exploration tools for Isabelle. The key implementation details are:

- A synthesis module responsible for the generation of conjectures and definitions using a constraint satisfaction solver.
- A normalisation module responsible for the normalisation of instantiations and the associated completion and termination techniques.
- An identification module responsible for the selection of important or relevant results (theorems and well-defined recursive functions).

Chapter 7

Evaluation

In this chapter, we evaluate IsaScheme. There are two main areas of evaluation that we consider. Firstly, we assess the theorems and definitions discovered by IsaScheme (§7.1, §7.2, §7.3 and §7.4). Secondly, we compare IsaScheme with other programs which perform mathematical theory-exploration (§7.6). We test two hypotheses:

- A theory-exploration framework based on schemes can be used for the problem of theorem discovery and concept invention.
- There exists a small number of schemes that consistently instantiate to a large number of theorems and/or concepts that mathematicians generally find to be interesting, while maintaining a relatively small quantity of uninteresting theorems and concepts.

There is a methodological problem with the evaluation of theorems and definitions produced by a theory-exploration system. The main difficulty is that such an evaluation requires judgments to be made about certain properties that are rather subjective and hard to define. Bearing this in mind, we conducted several case studies in the theory of natural numbers and the theory of lists to evaluate how similar were the results obtained with our method and implementation to those in the libraries of the Isabelle proof assistant. We performed a precision/recall analysis with Isabelle's libraries as reference to evaluate the quality of the theorems and definitions found by the `InventTheorems` and `InventDefinitions` algorithms respectively. To perform this analysis we used a fixed set of schemes which were not modified during the experiments. We also conducted experiments in theories not included in Isabelle to evaluate additional properties or to compare IsaScheme with similar theory-exploration programs. For the proof obligations (parameter $\mathcal{P}$ of the algorithms) we used the `induct_auto_tac` tactic (unless

stated otherwise). We kept track of elapsed time, conjectures and definitions synthesised, conjectures proved and not proved. For the evaluation we used a computer cluster where each theory-exploration was run in one GNU/Linux node with 2 dual core CPUs and 4GB of RAM memory. We also used Isabelle/2009-2 and IsaPlanner svn version 2723.

7.1 Natural Numbers

The evaluation of the `InventDefinitions` algorithm was performed with a theory consisting of one datatype for the naturals (see Appendix A). We used the following definitional scheme for the analysis

$$\left(\begin{array}{l} def\text{-}scheme\text{-}binary\ G\ H\ I\ J\ K\ L \equiv \\ \qquad \exists f.\ \forall x y z.\quad \bigwedge \left\{ \begin{array}{l} f\ G\ y = H\ y \\ f\ (I\ z\ x)\ y = J\ (K\ x\ y\ z)\ (f\ x\ (L\ z\ y)) \end{array} \right. \end{array} \right) \tag{7.1}$$

along with the scheme (7.2) and the terms of interest $\{0, suc, suc\ 0\}$ with which the algorithm described in section 4.6 automatically generated the closed terms

$$X_P = \left\{ \begin{array}{c} (\lambda x y.\ x), (\lambda x y.\ y), (\lambda x.\ suc), \\ (\lambda x y.\ suc\ x), (\lambda x y.\ 0), (\lambda x y.\ suc\ 0), \end{array} \right\}$$

$$X_D = \left\{ \begin{array}{c} (\lambda x.\ x), (\lambda x y.\ x), (\lambda x y.\ y), (\lambda x y z.\ x), (\lambda x y z.\ y), (\lambda x y z.\ z), \\ (\lambda x.\ 0), (\lambda x y.\ 0), (\lambda x.\ suc\ 0), 0, (\lambda x.\ suc), suc \end{array} \right\}.$$

Here, the set of closed terms X_D was used in the algorithm `InventDefinitions` and X_P was used in the recursive call to `InventTheorems`.

During this exploration round, IsaScheme found 16 new functions and two equivalent definitions for addition. However, the standard version of addition in A.1 of Appendix A was rejected because it was synthesised after $\oplus$ defined as:

$$\begin{aligned} 0 \oplus y &= y \\ suc\ x \oplus y &= x \oplus suc\ y. \end{aligned}$$

We further explored the theory by adding $\oplus$ to the sets X_P and X_D. During this exploration round, IsaScheme ended up with 31 functions. Multiplication $\otimes$ was found by IsaScheme in this exploration round, defined as:

$$\begin{aligned} 0 \otimes y &= 0 \\ suc\ x \otimes y &= y \oplus (x \otimes y). \end{aligned}$$

Again, we further explored the theory by adding $\oplus$ and $\otimes$ to the sets X_P and X_D. At this exploration round, IsaScheme ended up with 39 functions (including exponentiation) and proved 162 theorems leaving 3 unfalsified and unproved conjectures. For the theory of natural numbers, IsaScheme obtained a precision of 8% and a recall of 100%. The results for the evaluation of definitions are summarised in table 7.1. Note that the scheme-based approach for the generation of definitions provides a free-form incremental construction of (potentially infinitely many) recursive functions. Overly general definitional schemes and bigger sets of closed terms provide a wide range of possible instantiations and thus, definitions. In fact, we believe this was the reason for the low precision in the evaluation of the algorithm for this theory. Strategies to assess the relevance of definitions are required given the big search space during exploration. This is left as future work.

7.1.1 Naturals with Addition, Multiplication and Exponentiation

We evaluated the `InventTheorems` algorithm with a theory consisting of one datatype, of the naturals, and the usual recursive functions for addition, multiplication and exponentiation (which can be found in section A.1 of Appendix A). For the analysis of this theory we used the following scheme

$$\left(\begin{array}{l} \mathit{prop\text{-}scheme\text{-}binary}\ P\ Q\ R\ S\ T\ U \equiv \\ \qquad \forall x\,y\,z.\ P\ (Q\ x\ y)\ (R\ x\ z) = S\ (T\ x\ z)\ (U\ y\ z) \end{array} \right) \tag{7.2}$$

and the terms of interest $\{0, \mathit{suc}, \mathit{suc}\ 0, +, *, \hat{}\}$ with which the algorithm described in section 4.6 automatically generated the following closed terms:

$$X_P = \left\{ \begin{array}{c} (\lambda x\,y.\ x), (\lambda x\,y.\ y), (\lambda x.\ \mathit{suc}), (\lambda x\,y.\ \mathit{suc}\ x), \\ (\lambda x\,y.\ 0), (\lambda x\,y.\ \mathit{suc}\ 0), (\lambda x\,y.\ y + x), +, (\lambda x\,y.\ y * x), *, (\lambda x\,y.\ y \hat{} x), \hat{} \end{array} \right\}$$

IsaScheme produced a total of 16 theorems for the theory of naturals, with all 16 of them included in Isabelle's libraries. Isabelle contains 35 theorems about addition, multiplication and exponentiation giving a precision of 100% and a recall of 46%. The theorems discovered in the theory of natural numbers can be found in table D.1 of Appendix D and included, commutativity, associativity, distributivity of multiplication over addition, distributivity of exponentiation over multiplication, commuted versions of addition and multiplication, among others.

There are 19 theorems in Isabelle not synthesised by IsaScheme. However, we were surprised to see that all 19 theorems are normalised to *True* (including the theorems

that contained 4 variables) by the term rewrite system $\mathcal{R}$ in table D.1 of Appendix D. The theorems not synthesised are shown below:

$$(a*m)+m=(a+suc\ 0)*m \qquad a+(c+d)=(a+c)+d$$
$$m+a*m=(a+suc\ 0)*m \qquad (a+b)+c=(a+c)+b$$
$$m+m=((suc\ 0)+(suc\ 0))*m \qquad a*suc\ 0=a$$
$$(lx*ly)*(rx*ry)=lx*(ly*(rx*ry)) \qquad (suc\ 0)*a=a$$
$$lx*(rx*ry)=(lx*rx)*ry \qquad (x\hat{}q)*x=x\hat{}suc\ q$$
$$(lx*ly)*rx=(lx*rx)*ly \qquad x*(x\hat{}q)=x\hat{}suc\ q$$
$$x\hat{}(suc\ (suc\ 0)*n)=(x\hat{}n)*(x\hat{}n) \qquad x*x=x\hat{}(suc\ (suc\ 0))$$
$$x\hat{}(suc\ (suc\ (suc\ 0))*n)=x*((x\hat{}n)*(x\hat{}n)) \qquad x\hat{}suc\ 0=x$$
$$(lx*ly)*(rx*ry)=(lx*rx)*(ly*ry) \qquad (lx*ly)*(rx*ry)=rx*((lx*ly)*ry)$$
$$(a+b)+(c+d)=(a+c)+(b+d)$$

For instance, the theorem $(lx*ly)*(rx*ry)=(lx*rx)*(ly*ry)$ is reduced to $True$ by the theorems 11, 12 and 12 of table D.1 ($x*y=y*x$, $x*(y*z)=y*(x*z)$ and $(x*y)*z=x*(y*z)$) and the reflexivity theorem $(x=x)\equiv True$ in table 4.5 of section 4.5.2.

Precision-Recall	8%-100%	25%-21%
Constructors	Z, S	N, C
Function Symbols	$\oplus$,$\otimes$	
Elapsed Time (s)	24564	5546
Conjectures Synthesised	1991478	280008
Conjectures Filtered	1991313	279991
Proved-Not Proved	162-3	15-2
Definitions Synthesised	75059	11563
Definitions not Rejected	39	23

Table 7.1: Precision/recall analysis for definition synthesis with Isabelle's theory library as reference. The constructors are 0, *suc*, [] and # with labels Z, S, N and C respectively.

7.1.2 Naturals with Gödel's Recursor

Addition, multiplication and exponentiation of natural numbers can be encoded with Gödel's recursor:

$$\begin{aligned} rec\ 0\ y\ F &= y \\ rec\ (suc\ x)\ y\ F &= F\ x\ (rec\ x\ y\ F) \end{aligned}$$

For instance, addition, multiplication and exponentiation can be represented by the terms:

$$\begin{aligned} +_g &\equiv (\lambda xy.\ rec\ x\ y\ (\lambda uv.\ suc\ v)) \\ *_g &\equiv (\lambda xy.\ rec\ x\ 0\ (\lambda uv.\ y\ +_g\ v))) \\ \hat{}_g &\equiv (\lambda xy.\ rec\ y\ (suc\ 0)\ (\lambda uv.\ x\ *_g\ v))) \end{aligned}$$

Isabelle's naturals theory does not contain these definitions, so a precision/recall analysis was not possible in this theory. However, we wanted to see the theorems produced by the `InventTheorems` algorithm. For the experiment, we used the scheme (7.2) and the closed terms:

$$X_P = \left\{ \begin{array}{c} (\lambda xy.\ x), (\lambda xy.\ y), (\lambda x.\ suc), (\lambda xy.\ suc\ x), \\ (\lambda xy.\ 0), (\lambda xy.\ suc\ 0), +_g, *_g, \hat{}_g \end{array} \right\}$$

The `InventTheorems` algorithm discovered 17 theorems about addition, multiplication and exponentiation using Gödel's recursor. The theorems discovered can be found in table D.2 of Appendix D. The statistics are summarised in table 7.2.

Constructors	Z, S		Z, S, L, N
Function Symbols	$+$, $*$, $\hat{}$	$\lhd$, $\trianglelefteq$, $\rhd$, $\trianglerighteq$	$+$, Ma, Mi, No, H
Elapsed Time (s)	5845	978	4757
Conjectures Synthesised	111434	3912	259711
Conjectures Filtered	111417	3870	259687
Proved-Not Proved	17-0	42-0	23-1

Table 7.2: Results obtained in the theory of naturals with addition, multiplication and exponentiation encoded with Gödel's recursor; a theory about operators in set theory, and a theory about trees. The constructors are 0, *suc*, $Leaf$ and $Node$ with labels Z, S, L and N respectively. The functions are $+_g$, $*_g$, $\hat{}_g$, $\lhd$, $\trianglelefteq$, $\rhd$, $\trianglerighteq$, *max*, *mirror*, *nodes* and *height* with labels $+$, $*$, $\hat{}$, $\lhd$, $\trianglelefteq$, $\rhd$, $\trianglerighteq$, Ma, Mi, No and H respectively.

7.2 Excluding Defining Equations

The theorems discovered about addition, multiplication and exponentiation encoded with Gödel's recursor can be difficult to interpret. The reason is that these function symbols are rewritten in terms of *rec* during the normalisation process. This produces terms which are difficult to read and interpret. IsaScheme can be instructed to avoid using certain rewrite rules specified by the user. This experiment[1] shows the theorems obtained in a theory with 4 function symbols where the defining equations are not used during normalisation. The function symbols are defined as follows:

$$A \lhd R \equiv \{r \,|\, r \in R \wedge fst\ r \in A\}$$
$$A \lhd\!\!\!\cdot\ R \equiv \{r \,|\, r \in R \wedge fst\ r \notin A\}$$
$$R \rhd A \equiv \{r \,|\, r \in R \wedge snd\ r \in A\}$$
$$A \rhd\!\!\!\cdot\ R \equiv \{r \,|\, r \in R \wedge snd\ r \notin A\}$$

For the experiment, the proof method used was Isabelle's auto with all relevant definitions available during simplification. Again, the scheme (7.2) was used for the experiment in addition to the closed terms:

$$X_P = \left\{\ (\lambda xy.\ x), (\lambda xy.\ y), \cup, \lhd, \lhd\!\!\!\cdot, \rhd, \rhd\!\!\!\cdot\ \right\}.$$

Isabelle's theory libraries does not contain these definitions, so this theory has been excluded from the analysis. The `InventTheorems` algorithm discovered 42 theorems about the operators. The theorems synthesised can be found in Table D.3 of Appendix D.2. The statistics are summarised in table 7.2.

7.3 Lists

The evaluation of the `InventDefinitions` algorithm was performed with a theory consisting of one datatype for the lists. We used the scheme (7.2), the definitional scheme (7.1) and the terms of interest $\{[\,],\#\}$ with which IsaScheme automatically generated the following closed terms

$$X_P = \left\{\ (\lambda xy.\ x), (\lambda xy.\ y), (\lambda xy.\ [\,]), \#, (\lambda xy.\ y \,\#\, x)\ \right\}$$
$$X_D = \left\{ \begin{array}{c} (\lambda x.\ x), (\lambda xy.\ x), (\lambda xy.\ y), (\lambda xyz.\ x), (\lambda xyz.\ y), \\ (\lambda xyz.\ z), (\lambda x.\ [\,]), [\,], (\lambda x.\ \#), \#, (\lambda xy.\ y \,\#\, x) \end{array} \right\}.$$

[1] The experiment involved a collaboration with Gudmund Grov for the AI4FM project and here we merely describe the results we obtained.

IsaScheme produced 24 definitions for the theory of lists with 6 of them included in Isabelle's libraries. Isabelle contains 29 recursive functions in its `List` theory giving a precision of 25% and a recall of 21%. The definitions synthesised by IsaScheme were append, (binary version of) head (*hd*), map, (binary version of) last (*last*), tail recursive reverse (*qrev*) and replicate (*replicate*), among others. A representative selection of definitions synthesised are shown in section A.2 of Appendix A.

The low recall for the list theory was primarily due to the fact that the definitional scheme used was only able to synthesise binary functions and not unary or ternary ones. This could have been addressed easily by considering definitional schemes producing unary and ternary functions, albeit at the expense of computational time (see section 4.2). Another reason for the low recall was that, in order to prevent a combinatorial explosion, we also restricted the set of closed terms by using only the list constructors as the terms of interest. Hence, definitions requiring the constructors for the naturals (0 and *suc*) or conditionals (if, *nat_case* and *list_case*) such as *drop*, *take*, *count*, *zip*, etc. were not synthesised. We experimented with unary and ternary definitional schemes together with the constructors for the naturals and conditionals. However, we reached the maximum allowable time for jobs in the computer cluster of 48 hours. Analogous to section 7.1, overly general definitional schemes and bigger sets of closed terms provide a wide range of possible instantiations and thus, definitions. Again, we believe this was the reason for the low precision in the evaluation of the algorithm for this theory. Strategies to assess the relevance of definitions are required given the big search space during exploration. This is left as future work. For space reasons we cannot give a presentation of the theories or the theorems and definitions found. However, formal theory documents in human-readable Isabelle/Isar notation and all results described in this thesis are available online[2].

To assess the quality of theorems about lists found by IsaScheme, we have considered three different test sets. Firstly, we performed a precision/recall analysis with Isabelle's list theory considering only theorems about append (@), reverse (*rev*), map (*map*), length (*len*), fold-left ($foldl$) and fold-right ($foldr$) (see section 7.3.1). Secondly, we have aimed to compare our system with similar theory-exploration programs such as IsaCoSy or MATHsAiD. They have been applied in different theories about append, reverse, tail recursive reverse, length, addition and equality. To compare our system with the aforementioned programs we have tested IsaScheme in two theories: (1) theory about lists (and naturals) with append, reverse, length, addition and equality

[2] http://dream.inf.ed.ac.uk/projects/isascheme/eswa2

(see section 7.3.2) and (2) theory about lists with append, reverse and tail recursive reverse (see section 7.3.3).

7.3.1 Lists with Append, Reverse, Map, Length, Fold-left and Fold-right

We evaluate the `InventTheorems` algorithm using two different theories of lists. The definitions of the functions used in these theories can be found in section A.2 of Appendix A. The first theory consisted of append (@), reverse (*rev*), map (*map*) and length (*len*). For the analysis of this theory we used the scheme (7.2) with the terms of interest $\{[], \#, @, rev, map, len\}$ with which IsaScheme automatically generated the following closed terms

$$X_P = \left\{ \begin{array}{c} (\lambda xy.\ x), (\lambda xy.\ y), (\lambda x.\ []), \#, @, (\lambda xy.\ y\ @\ x), (\lambda xy.\ rev\ x) \\ (\lambda xy.\ rev\ y), (\lambda xy.\ len\ x), (\lambda xy.\ len\ y), map, (\lambda xy.\ map\ y\ x) \end{array} \right\}.$$

IsaScheme produced a total of 9 theorems for this theory, including all 7 theorems about @, *rev*, *map* and *len* (see table D.4) in Isabelle's libraries. This gives a precision of 77% and a recall of 100%. The extra theorems synthesised were $len\ (z\ @\ x) = len\ (x\ @\ z)$ and $len\ (map\ y\ x) = len\ x$.

The second theory analysed consisted of the functions append (@), fold-left ($foldl$) and fold-right ($foldr$). To handle ternary operators such as $foldl$ and $foldr$, we used another scheme

$$\left(\begin{array}{l} prop\text{-}scheme\text{-}ternary\ P\ Q\ R\ S\ T\ U\ V \equiv \\ \quad \forall wxyz.\ P\ (V\ z\ x\ y)\ (Q\ z\ x\ y)\ (R\ w\ x\ z) = \\ \quad\quad S\ z\ (T\ y\ x\ z)\ (U\ z\ y\ w) \end{array} \right) \tag{7.3}$$

with the following closed terms:

$$X_P = \left\{ \begin{array}{c} (\lambda xyz.\ x), (\lambda xyz.\ y), (\lambda xyz.\ z), (\lambda xyz.\ []), (\lambda xyz.\ \#) \\ (\lambda x.\ @), (\lambda xyz.\ @), (\lambda xy.\ y\ @\ x), foldl, foldr \end{array} \right\}.$$

IsaScheme produced a total of 4 theorems for the theory of lists producing 2 out of 4 theorems about append, fold-left and fold-right included in Isabelle's libraries. Associativity of append $(x\ @\ y)\ @\ z = x\ @\ (y\ @\ z)$ and $x\ @\ [] = x$ were not synthesised by IsaScheme. Interestingly, append was redefined in terms of fold-right and the list constructor # by $x\ @\ z = foldr\ \#\ x\ z$. This caused associativity of append to be subsumed by the more general version $foldr\ z\ (foldr\ \#\ x\ y)\ w = foldr\ z\ x\ (foldr\ z\ y\ w)$

and x @ $[] = x$ to be subsumed by $foldr$ # z $[] = z$ (after normalisation w.r.t. the rewrite system $\mathcal{R}$). In this theory, there was a rather high number of unfalsified and unproved conjectures (129). The reason for this is that the type information in variables for these conjectures is more complex than Quickcheck and Nitpick can manage. A small random sample of these conjectures was taken, and in each case, a counterexample was quite easily found by hand. For instance, one of these unfalsified (and unproved) conjectures was $foldr\ z\ x\ \# = \#$, which can be falsified with the witness $\sigma = \{z \mapsto (\lambda a\,b\,c\,d.\ d), x \mapsto [a]\}$.

In total, IsaScheme produced 13 theorems for the two theories for lists analysed producing all 9 theorems about append, list reverse, map, right-fold and left-fold included in Isabelle's libraries. This gives a precision of 70% and a recall of 100%. Table 7.3 summarises the statistics for the theories analysed.

Precision-Recall	70%-46%	77%-100%	50%-50%
Constructors	Z, S	Z, S, N, C	N, C
Function Symbols	+, *, ^	A, R, M, L	A, FL, FR
Elapsed Time (s)	1663	1986	8726
Conjectures Synthesised	78957	47142	13213
Conjectures Filtered	78934	47133	13209
Proved-Not Proved	23-0	9-0	4-129

Table 7.3: Precision/recall analysis with Isabelle's theory library as reference. The constructors are 0, *suc*, $[]$ and # with labels Z, S, N and C respectively. The functions are $+$, $*$, ^, @, *rev*, *len*, *map*, $foldl$ and $foldr$ with labels +, *, ^, A, R, L, M, FL and FR respectively.

7.3.2 Lists with Append, Reverse, Length, Addition and Equality

Isabelle/HOL contains an object level equality ($=$) which can be used to equate formulae and even functions over formulae [55]. Object level equality can also be used as an ordinary function symbol during theory-exploration in IsaScheme. We experimented with a theory of lists with append (@), reverse (*rev*), length (*len*), addition ($+$) and object level equality ($=$). For the experiment we use the scheme 7.2 and the following closed terms (generated automatically by IsaScheme from the terms of interest

$\{[],\#,@,rev,len,+,=\}$)

$$X_P = \left\{ \begin{array}{c} (\lambda xy.\ x),(\lambda xy.\ y),(\lambda x.\ []),\#,(\lambda xy.\ y\ \#\ x),@,(\lambda xy.\ y\ @\ x),(\lambda xy.\ rev\ x) \\ (\lambda xy.\ rev\ y),(\lambda xy.\ len\ x),(\lambda xy.\ len\ y),+,(\lambda xy.\ y+x) \\ (\lambda xy.\ suc\ y),(\lambda xy.\ suc\ x),(\lambda xy.\ 0),=,(\lambda xy.\ y=x) \end{array} \right\}.$$

The `InventTheorems` algorithm discovered 24 theorems about functions $+$, *len*, *@*, *rev* and $=$. The theorems synthesised can be found in table D.7 of Appendix D and the statistics are summarised in table 7.4.

7.3.3 Lists with Append, Reverse and Tail Recursive Reverse

Automatically proving properties of tail-recursive function definitions by induction is known to be challenging [31, 38, 32]. This experiment, in a theory of lists with append (*@*), reverse (*rev*) and tail-recursive reverse (*qrev*), shows how theory-exploration can be a useful technique to discover important properties about such functions. In particular, IsaScheme conjectures and proves a property relating list reverse with its tail-recursive version ($qrev\ z\ x = (rev\ z)\ @\ x$; see Theorem 5 in Table D.6 of Appendix D). For the experiment we use the scheme 7.2 and the following closed terms

$$X_P = \left\{ \begin{array}{c} (\lambda xy.\ x),(\lambda xy.\ y),(\lambda x.\ []),\#,(\lambda xy.\ y\ \#\ x),@,(\lambda xy.\ y\ @\ x), \\ (\lambda xy.\ rev\ x),(\lambda xy.\ rev\ y),qrev,(\lambda xy.\ qrev\ y\ x) \end{array} \right\}.$$

The `InventTheorems` algorithm discovered 5 theorems about functions *@*, *rev* and *qrev*. The theorems synthesised can be found in table D.6 of Appendix D and the statistics are summarised in table 7.4.

7.4 Trees

The IsaCoSy program was evaluated in a theory of trees which is why we have decided to test IsaScheme in the same theory. The theory of trees consisted of one datatype each for trees and the naturals, together with the functions $+$, *max*, *mirror*, *nodes* and *height* (see Appendix A). For the analysis we used the scheme 7.2 and the following closed terms (generated automatically by IsaScheme from the terms of interest $\{0, suc, Leaf, Node, +, max, mirror, nodes, height\}$)

$$X = \left\{ \begin{array}{c} (\lambda xy.\ x),(\lambda xy.\ y),(\lambda x.\ suc),(\lambda xy.\ suc\ x),(\lambda xy.\ 0),+,(\lambda xy.\ y+x), \\ max,(\lambda xy.\ max\ y\ x)(\lambda xy.\ mirror\ x),(\lambda xy.\ mirror\ y),(\lambda xy.\ height\ x), \\ (\lambda xy.\ height\ y),(\lambda xy.\ nodes\ x),(\lambda xy.\ nodes\ y) \end{array} \right\}.$$

Constructors	Z, S, N, C	N, C
Function Symbols	$+$, L, @, R, E	@, R, Q
Elapsed Time (s)	3068	4678
Conjectures Synthesised	121619	160454
Conjectures Filtered	121595	160449
Proved-Not Proved	24-0	5-0

Table 7.4: Results obtained in the theory of lists (and naturals) with addition, length, append, reverse, tail recursive reverse and object level equality. The constructors are 0, *suc*, $[]$ and # with labels Z, S, N and C respectively. The functions are $+$, *len*, @, *rev*, *qrev* and $=$ with labels $+$, L, @, R, Q and E respectively.

The `InventTheorems` algorithm discovered 23 theorems about functions $+$, *max*, *mirror*, *nodes* and *height* and one unfalsified and unproved conjecture (*nodes* $x =$ *max* (*height* x) (*nodes* x)). The theorems synthesised can be found in table D.8 of Appendix D and the statistics are summarised in table 7.2.

7.5 Analysis of Precision/Recall

We have compared the theories about naturals and lists produced by IsaScheme with those in Isabelle's libraries. The exploration of those theories was performed automatically. IsaScheme not only synthesised the conjectures and definitions without user intervention, it also proved the theorems automatically using the proof technique described in chapter 5. This verifies the hypothesis that the scheme-based approach can be automated to produce conjectures and definitions.

To assess the quality of the theorems found by IsaScheme we performed a precision/recall analysis with Isabelle's libraries. IsaScheme produced many interesting theorems resulting in high precision of 100% for the natural numbers and 70% for the theory of lists. IsaScheme finds all the theorems for the theory of lists resulting in a recall of 100% but it does not find all the theorems for the naturals, obtaining only 46%. Surprisingly, all theorems not found were subsumed by the theorems IsaScheme synthesised, including those with 4 variables. IsaScheme was also evaluated in theories not included in the libraries of Isabelle. In these theories, IsaScheme was able to prove inductive theorems that would otherwise require sophisticated proof techniques

such as rippling [11] and proof planning critics [30] (see section 7.6.3). An important feature of IsaScheme is that it constructs a terminating and potentially convergent rewrite system which can be used directly by provers that perform rewriting such as Isabelle's Simplifier or rippling. This rewrite system has been exploited by the proof technique described in chapter 5 proving many challenging theorems.

We also evaluated the quality of the definitions synthesised by IsaScheme. For this we performed a precision/recall analysis with the functions found in Isabelle's libraries. IsaScheme produced all the definitions for the theory of naturals after 3 exploration rounds producing a recall of 100%. However, IsaScheme produced a low recall of definitions for the theory of lists obtaining only 21%. The main culprit was the definitional scheme used, as it was only able to synthesize binary functions and not unary or ternary ones. IsaScheme produced a low precision for definitions in the theory of natural numbers and lists, obtaining 8% for the natural numbers and 26% for the theory of lists. Strategies to assess the relevance of definitions are, however, an interesting further work for IsaScheme given the big search space during exploration.

7.6 Related Work

In chapter 2 we gave an overview of some of the research literature related to our work. In this section we briefly recapitulate that material, and provide more detail of certain mathematical theory-exploration systems, in order to compare them with IsaScheme.

7.6.1 Theorema

Other than IsaScheme, Theorema is the only system performing the exploration of mathematical theories based on schemes [8]. However, the user needs to perform all schematic substitutions manually as Theorema does not instantiate the schemes automatically from a set of terms. As an illustration, we take a small extract of the exploration performed in [20] for a theory of natural numbers. The authors used a set of 23 schemes. The schemes included formulae to generate definitions such as addition and multiplication, conjectures such as associativity and commutativity, etc. Some of the schemes are depicted in figure 7.1. Note that the schemes have been adapted into a typed framework to unify the presentation. Moreover, definitional schemes in Theorema does not have existentially quantified variables for the new function symbols (Theorema ambiguously uses free variables to give name at new function symbols).

However, here we assume definitional schemes as defined in section 4.1. The exploration started when the authors specified the substitution $\{G \mapsto id,\ H \mapsto suc\}$ on scheme *is-rec-nat-binary-fct-1l*. This substitution led to addition of natural numbers and they assigned the syntax '$+$' to this new function. Then the authors manually provided the substitutions $\{P \mapsto +\}$ and $\{P \mapsto +,\ Q \mapsto 0\}$ on *is-semigroup* and *is-monoid* respectively. This in turn generated the associativity property of addition and the neutral element property (which is already in the definition of addition).

$$\left(\begin{array}{l} \textit{is-rec-nat-binary-fct-1r}\ G\ H\ \equiv \\ \qquad \exists f.\ \forall xy.\ \bigwedge\left\{\begin{array}{rcl} f\ x\ 0 & = & G\ x \\ f\ x\ (\textit{suc}\ y) & = & H\ (f\ x\ y) \end{array}\right. \end{array}\right)$$

$$\left(\begin{array}{l} \textit{is-group}\ P\ Q\ R \equiv \\ \qquad \forall x.\ \bigwedge\left\{\begin{array}{l} \textit{is-monoid}\ P\ Q \\ P\ x\ (R\ x) = Q \end{array}\right. \end{array}\right)$$

$$\left(\begin{array}{l} \textit{is-semigroup}\ P \equiv \\ \qquad \forall xyz.\ P\ x\ (P\ y\ z) = P\ (P\ x\ y)\ z \end{array}\right)$$

$$\left(\begin{array}{l} \textit{is-monoid}\ P\ Q \equiv \\ \qquad \forall x.\ P\ x\ Q = x \end{array}\right)$$

Figure 7.1: Some schemes used by Theorema to generate definitions in an exploration of the naturals.

The same sequence of exploration steps are generated by IsaScheme but without user intervention. The user only needs to provide the terms of interest 0 and *suc*. One difference is that IsaScheme additionally generates three new functions depicted in figure 7.2. For each function, IsaScheme then instantiates the schemes *is-group*, *is-semigroup* and *is-monoid* and finds the theorems: $x+(y+z) = (x+y)+z$, $x \oplus (y \oplus z) = (x \oplus y) \oplus z$ and $x \ominus x = 0$.

In theorema, the user also needs to discharge the proof obligations interactively [20] as opposed to IsaScheme. Another important difference is that ensuring the soundness of definitions is left to the user in Theorema. In IsaScheme, which uses Isabelle's LCF-methodology, definitions are sound by construction [39].

$$\begin{aligned} x \oslash 0 &= suc\ x \\ x \oslash (suc\ y) &= 0, \\ x \ominus 0 &= x \\ x \ominus (suc\ y) &= 0, \\ x \oplus 0 &= suc\ x \\ x \oplus (suc\ y) &= suc\ (x \oplus y) \end{aligned}$$

Figure 7.2: Definitions invented by IsaScheme using the *is-rec-nat-binary-fct-1r* scheme of figure 7.1.

7.6.2 MATHsAiD

The MATHsAiD program was intended for use by research mathematicians and was designed to produce interesting theorems from the mathematician's point of view [48]. MATHsAiD starts with an axiomatic description of a theory; hypotheses and terms of interest are then generated, forward reasoning is then applied to produce logical consequences of the hypotheses and then a filtering process is carried out according to a number of interestingness measures.

MATHsAiD has been applied to inductive theories such as natural numbers and lists. The synthesis process of inductive conjectures starts with the construction of an 'interesting' proposition $P\ n$ that holds for the case TWO (e.g. $suc\ (suc\ 0)$ for naturals). Then the appropriate induction scheme is used to test if the 'base' case is true and, if so, forward reasoning is conducted to prove the 'step' case.

MATHsAiD also uses heuristics to generate 'routine' theorems. For example, given a binary operator, MATHsAiD tries to prove the commutativity and associativity properties; and given a pair of these operators, MATHsAiD tries to prove the distributivity property. In the theory of naturals, MATHsAiD discovers 5 theorems that IsaScheme did not generate:

$$\begin{aligned} a + suc\ 0 &= suc\ a \\ suc\ 0 + b &= suc\ b \\ a * suc\ 0 &= a \\ (suc\ 0) * b &= b \\ a + (b * c) &= (c * b) + a \end{aligned}$$

IsaScheme does not produce them because they are subsumed during the normalisation

process. For example, $suc\ 0+b = suc\ b$ is an instance of the more general theorem $suc\ x+y = suc\ (x+y)$ which both systems discover. However, MATHsAiD uses a different heuristic to decide what is worth recording. One of the heuristics is to impose a limit in the size of the right-hand side of equations, computed as $M+size(TWO)+2$. Here M is the size of the left-hand side of the equation, $size(TWO)$ is the size of the case TWO and the size of a term is defined as:

$$size(t) := \begin{cases} 0 & \text{if } t \text{ is a variable or a constant} \\ 1+\sum_{i=1}^{n} size(s_i) & \text{otherwise if } t \text{ is an application } f(s_1,\ldots,s_n) \end{cases}$$

In the theory of lists, MATHsAiD discovers 3 theorems that IsaScheme did not generate, again because reduction filtered them out:

$$\begin{aligned} (a\ \#\ [])\ @\ y &= a\ \#\ y \\ rev\ (a\ \#\ []) &= a\ \#\ [] \\ len\ (a\ \#\ []) &= suc\ 0 \end{aligned}$$

In terms of speed, IsaScheme is significantly slower than MATHsAiD. While IsaScheme generates the theorems for the naturals[3] in 1809 seconds and the theorems for lists in 3068 seconds, MATHsAiD generates the theorems for the naturals in no more than 120 seconds and just 253 seconds to generate the theorems for lists. This is not surprising, as the deductive approach is heavily used by MATHsAiD. Here, every output is known to be a theorem and thus counter-example checking is unnecessary. MATHsAiD also uses a number of heuristics encoded to generate 'routine' theorems such as commutativity, associativity and distributivity. This evidently reduces exploration times, as the generation of these theorems does not require search.

7.6.3 IsaCoSy

IsaCoSy is a theory-exploration system for Isabelle/ IsaPlanner [33]. It generates conjectures in a bottom-up fashion from the signature of an inductive theory. The synthesis process is accompanied by automatic counter-example checking and a proof attempt with the inductive prover IsaPlanner in case no counter-example is found by Quickcheck.

Starting with a small term of some specified size, IsaCoSy builds larger ones incrementally using the function symbols in the signature of a given theory. All theorems

[3] IsaScheme's exploration of the naturals also included exponentiation, which was excluded from MATHsAiD's exploration of the naturals.

found are then used as constraints (in a constraint language developed for IsaCoSy) for the synthesis process generating only irreducible terms w.r.t. the discovered theorems. The process continues iteratively until a given maximum size is reached.

Similar to MATHsAiD, IsaCoSy uses heuristics to restrict the synthesis of equations. Synthesised equations are required to have the left-hand side be of larger or equal size than the right-hand side. For example, $rev\ (rev\ y) = y$ is a valid equation, but $y = rev\ (rev\ y)$ is not. Also, the set of variables appearing in the right-hand side of the equation must be a subset of the variables on the left-hand side. For instance, $(x+y = x+z) = (y = z)$ is a valid equation but, $(suc\ x = x+y) = (suc\ z = y+z)$ is not. Optionally, IsaCoSy also tries to prove (if the option is switched on) associativity, commutativity and commuted versions of definitions. For example, for addition IsaCoSy eagerly synthesises:

$$\begin{aligned}(x+y)+z &= x+(y+z)\\ x+y &= y+x\\ x+0 &= x\\ x+suc\ y &= suc\ (x+y)\end{aligned}$$

The main difference between IsaScheme and IsaCoSy is that IsaCoSy considers all (irreducible) terms (without λ-abstractions) as candidate conjectures where IsaScheme considers only a restricted set (modulo $\mathcal{R}$) specified by the schemes. This restricted set of conjectures avoids the need for a sophisticated constraint language and provides an upper bound on the number of instantiations synthesised (Theorem 1 of section 4.2). In the theory of naturals, IsaCoSy discovers 8 different versions of distributivity of multiplication over addition that IsaScheme does not generate.

$$\begin{array}{ll}(a*b)+(c*b) = (a+c)*b & (a*b)+(c*a) = (b+c)*a\\ (a*b)+(c*a) = (c+b)*a & (a*b)+(c*b) = (c+a)*b\\ (a*b)+(a*c) = (b+c)*a & (a*b)+(a*c) = (c+b)*a\\ (a*b)+(b*c) = (a+c)*b & (a*b)+(b*c) = (c+a)*b\end{array}$$

However, all of them are subsumed after normalisation w.r.t. the rewrite system $\mathcal{R}$ constructed by IsaScheme. IsaCoSy also generated 1 theorem ($(0\hat{}a)*a = 0$) and 2 unfalsified and unproved conjectures ($0\hat{}(a\hat{}a) = 0$ and $0\hat{}((suc\ a)\hat{}b) = 0$) that escaped the scheme used during synthesis by IsaScheme. These were not subsumed by the theorems synthesised by IsaScheme, but they could have been generated by either a more elaborate scheme or a set of closed terms containing, for instance, the terms

$\lambda xy.\ 0\hat{}y$ and $\lambda xy.\ (suc\ x)\hat{}y$ (in addition to the set of closed terms used during the experiment in section 7.1.1). IsaCoSy was also evaluated on a theory about append, reverse and length. In this theory, IsaCoSy discovers 12 extra theorems:

$$rev\ (a\ @\ rev\ b) = b\ @\ rev\ a \qquad rev\ a\ @\ rev\ b = rev\ (b\ @\ a)$$
$$rev\ (rev\ a\ @\ rev\ b) = b\ @\ a \qquad rev\ (a\ @\ [b]) = b\ \#\ rev\ a$$
$$rev\ (rev\ a\ @\ [b]) = b\ \#\ a \qquad rev\ (a\ @\ (b\ @\ rev\ c)) = c\ @\ rev\ (a\ @\ b)$$
$$rev\ (a\ @\ (b\ \#\ rev\ c)) = c\ @\ (b\ \#\ rev\ a) \qquad a\ @\ rev\ (rev\ b\ @\ c) = a\ @\ (rev\ c\ @\ b)$$
$$rev\ a\ @\ (rev\ b\ @\ c) = rev\ (b\ @\ a)\ @\ c \qquad rev\ a\ @\ (b\ \#\ rev\ c) = rev\ (c\ @\ (b\ \#\ a))$$
$$rev\ (rev\ a\ @\ b)\ @\ c = rev\ b\ @\ (a\ @\ c) \qquad rev\ (rev\ a\ @\ (b\ \#\ rev\ c)) = c\ @\ (b\ \#\ a)$$

and 2 unproved conjectures:

$$\begin{aligned} rev\ (rev\ a\ @\ b) &= rev\ b\ @\ a \\ rev\ (rev\ a\ @\ (b\ \#\ c)) &= rev\ c\ @\ (b\ \#\ a). \end{aligned}$$

Again, all of the theorems, including the conjectures not proved by IsaCoSy, are normalised to *True* by the rewrite system $\mathcal{R}$ IsaScheme synthesised. IsaCoSy was also used to synthesise properties about tail-recursive reverse in a theory of lists with append (@), reverse (*rev*) and tail-recursive reverse (*qrev*). There was a big difference between the number of theorems obtained by IsaCoSy and the number obtained by IsaScheme in this theory. While IsaCoSy generated 24 theorems and 45 unfalsified and unproved conjectures[4], IsaScheme generated only 5 theorems (see table D.6 in Appendix D). Interestingly, these 5 theorems subsumed all 24 theorems IsaCoSy synthesise including the 45 unfalsified and unproved conjectures. The reason was that IsaScheme managed to synthesise a convergent rewrite system $\mathcal{R}$ with enough simplification power to subsume the terms IsaCoSy synthesised.

IsaScheme also exploits associative-commutative (AC) operators using ordered rewriting to avoid AC variations of the same instantiation. A particular example where IsaScheme's AC rewriting technique successfully filtered AC variations was the aforementioned versions of distributivity of multiplication over addition that IsaCoSy produces.

A notable difference is that IsaScheme is capable of synthesising higher-order conjectures or definitions containing λ-terms. IsaCoSy is also able to synthesise higher-order conjectures but it cannot synthesise terms containing λ-terms. Hence, IsaCoSy is unable to synthesise the theorems depicted in table D.2 using Gödel's recursor or the

[4] See http://dream.inf.ed.ac.uk/projects/lemmadiscovery/results/List_rev_qrev.txt for details.

theorems depicted in table D.8 using case statements (e.g. the term $case\ y\ of\ 0 \Rightarrow suc\ x \mid suc\ z \Rightarrow suc\ (max\ z\ x)$ is represented internally in Isabelle by $case\_nat\ (suc\ x)\ (\lambda z.\, suc\ (max\ z\ x))\ y$).

IsaScheme can also use any tactic given by a user in standard Isabelle/Isar notation as a parameter of the system, including inductive tactics such as the one presented in chapter 5. This capability of taking the proof strategy as a parameter of IsaScheme was exploited with the theory of section 7.2 where the proof method specified was Isabelle's auto with all relevant definitions available during simplification. Contrary to this, IsaCoSy only uses a fixed prover, which in this case is IsaPlanner.

The main advance made by IsaScheme is the use of Knuth-Bendix completion and termination checking to orient the resulting equational theorems to form a rewrite system. The empirical results show that for the theory of lists, these rewrite systems result in fewer theorems that prove all of the theorems in the theory produced by IsaCoSy. In the theory of naturals, IsaCoSy generated one theorem and two unfalsified conjectures that escaped the scheme and the closed terms used during exploration, but they could have been generated and proved by either a more elaborate scheme or a set of closed terms containing two more terms.

7.6.4 HR

HR is a theory-exploration system which uses an example-driven approach for theory-formation [15]. It uses MACE [52] to build models from examples and also to identify counter-examples. The resolution prover Otter [49] is used for the proof obligations. The process of concept invention is carried out from old concepts starting with the concepts provided by MACE at the initial stage. These concepts, stored as data-tables of examples rather than definitions, are passed through a set of production rules whose purpose is to manipulate and generate new data-tables, thus generating new concepts. The conjecture synthesis process is built on top of concept formation. HR takes the concepts obtained by the production rules and forms conjectures about them. There are different types of conjectures HR can make, e.g. *equivalence* conjectures which amounts to finding two concepts and stating that their definitions are equivalent, *implication* conjectures are statements relating two concepts by stating that the first is a specialization of the second (all examples of the first will be examples of the second), etc.

One difference between HR and IsaScheme is the way concept-formation is per-

formed. Concept-formation is driven as a heuristic search in HR: less interesting concepts are developed after more interesting ones. There are heuristics to measure different aspects of each concept. IsaScheme does not differentiate between definitions; i.e. all definitions are equally important. Strategies to assess the relevance of definitions are, however, an interesting further work for IsaScheme given the big search space during exploration. One strategy could be the use of schemes as specifications for the new definition. With this technique, the user can apply a set of schemes which should be satisfied by each new concept invented. For example, new binary operators should have the associativity and/or commutativity properties.

Another difference is that HR uses the resolution prover Otter for the proof obligations as opposed to IsaScheme which uses any (Isabelle) tactic specified by the user. This allows IsaScheme to be used in theories in which HR's prover would have problems, such as inductive ones (see [15] pg. 221).

HR uses different measures to assess the interestingness of concepts and conjectures. Many of them are configured by the user (see [15] pg. 203) making it difficult to use. One heuristic noticeably similar to IsaScheme's normalisation technique is surprisingness. This heuristic avoids making tautologies of a particular type. For example, for any A and p, the conjecture

$$\neg(\neg(p\ A)) \leftrightarrow p\ A$$

is always going to be true. HR rejects this tautology by using an ad-hoc technique called *forbidden paths* (see [15] pg. 111). Forbidden paths are construction path segments (supplied to HR prior to theory-formation) HR is not allowed to take. For example, it is not allowed to follow a negation step with another negation step, hence avoiding the above tautology.

IsaScheme uses a more powerful and general technique based on normalisation w.r.t. a rewrite system $\mathcal{R}$. In IsaScheme, the above tautology is stored in $\mathcal{R}$ as a rewrite rule:

$$\neg(\neg(P)) \rightarrow P$$

which is used to normalise new conjectures and definitions. Moreover, the defining equations of new definitions and the lemmata discovered during the exploration of the theory are used to avoid further redundancies such as the one originated with the above tautology.

7.7 Summary

We put the work presented in this thesis into context by comparing IsaScheme with different theory-exploration programs. Other than IsaScheme, Theorema is the only system performing theory-exploration based on schemes. However, IsaScheme performs more autonomously than Theorema, as the latter requires the user to provide the appropriate substitutions manually. Moreover, ensuring the soundness of definitions is left to the user in Theorema. In IsaScheme, which uses Isabelles LCF-methodology, definitions are sound by construction.

A key strength of IsaScheme is its reduction technique. IsaScheme strives to maintain nice properties on the rewrite system $\mathcal{R}$ such as termination or convergence. This helps prune the search space of unwanted redundant definitions and theorems resulting in fewer theorems being synthesised with more simplification power than those generated by other systems. IsaCoSy also uses irreducibility as a measure of worth. However, IsaCoSy's constraint algorithm is not concerned with termination or convergence of the equations it synthesises. An example where this difference is crucial is in the theory of lists with append, reverse and tail-recursive reverse. In this theory, IsaScheme generated five theorems that subsumed the 24 theorems and 45 unproved conjectures that IsaCoSy generated.

The use of schemes to determine the shape of instantiations, will occasionally allow IsaScheme to miss some theorems. This was the case where IsaCoSy generated 1 theorem ($(0\hat{}a) * a = 0$) and 2 unfalsified and unproved conjectures ($0\hat{}(a\hat{}a) = 0$ and $0\hat{}((suc\ a)\hat{}b) = 0$) that escaped the scheme used in the evaluation of IsaScheme.

Chapter 8

Conclusions

This research was focused on the process of inventing mathematical concepts and inventing (and verifying) conjectures about those concepts. Our primary aim has been to improve upon automation of the proposed approach [6] to theory-formation 'scheme-based mathematical theory-exploration'. In the Theorema system the user had to provide the appropriate substitutions manually (Theorema cannot perform the possible instantiations automatically) and the proof obligations were in part 'pen-and-paper'. We also aimed to provide a model for theory-exploration that worked over a range of mathematical domains. To conclude our discussion of this work, in 8.1 we analyze whether these aims have been achieved. In 8.2 we propose directions for further work and in 8.3 we look again at the contributions this project makes to the state of the art in mathematical theory-exploration. A final summary is provided in section 8.4.

8.1 Have we achieved our aims?

The hypotheses of the project we proposed in chapter 1 were (i) a theory-exploration framework based on schemes can be used for the problem of theorem discovery and concept invention and (ii) there exists a small number of schemes that consistently instantiate to a large number of theorems and/or concepts that mathematicians generally find to be interesting, while maintaining a relatively small quantity of uninteresting theorems and concepts.

The hypotheses can be analysed in two parts. Firstly, the model for theory-exploration is able to synthesise important definitions. For evidence supporting this we refer back to the experimental results of table 7.1 in chapter 7. IsaScheme produced all the definitions for the theory of naturals after 3 exploration rounds producing a recall of 100%.

However, IsaScheme produced a low recall of definitions for the theory of lists obtaining only 21%. The culprit was the definitional scheme used, as it was only able to synthesize binary functions and not unary or ternary ones. This could have been addressed easily by considering definitional schemes producing unary and ternary functions, albeit at the expense of computational time (see section 4.2). IsaScheme produced a low precision for definitions in both theories, obtaining 8% for the natural numbers and 26% for the theory of lists. Strategies to assess the relevance of definitions are, however, interesting further work for IsaScheme, given the big search space during exploration. One strategy could be the use of schemes as specifications for the new definition. With this technique, the user can apply a set of schemes which should be satisfied by each new concept invented. For example, new binary operators should have the associativity and/or commutativity properties.

Secondly, the model for theory-exploration developed is also capable of finding many useful and interesting theorems. For evidence supporting this, we refer back to the experimental results obtained in the different theories explored in chapter 7 (see tables 7.3, 7.2 and 7.4). IsaScheme produces many interesting theorems resulting in high precision of 100% for the natural numbers and 70% for the theory of lists. IsaScheme finds all the theorems for the theory of lists resulting in a recall of 100% but it does not find all the theorems for the naturals, obtaining only 46%. Surprisingly, all theorems about the naturals not found were subsumed by the theorems IsaScheme synthesised, including those with 4 variables. IsaScheme was also evaluated in theories not included in the libraries of Isabelle. In these theories, IsaScheme was able to prove inductive theorems that would otherwise require sophisticated proof techniques such as rippling [11] and proof planning critics [30]. An important feature of IsaScheme is that it constructs a terminating and potentially convergent rewrite system which can be used directly by provers that perform rewriting such as Isabelle's simplifier or rippling. In chapter 5 we described a proof technique that exploits this terminating rewrite system obtained by IsaScheme and demonstrated the ability to prove challenging theorems.

We hope that the evidence we have provided is sufficient to convince the reader of the validity of our hypotheses.

8.2 Limitations and Further Work

An important limitation that is unavoidable for current theory-formation systems is their combinatorial complexity. The scheme-based approach used by IsaScheme pro-

vides a generic mechanism for the exploration of any mathematical theory where the symbols (or closed terms built from them) in the theory's signature and the variables within the schemes could be unified. This free-form theory-exploration could lead to a substantial number of instantiations that need to be processed (see section 4.2) and it is particularly true with large numbers of constructors and function symbols. Theorem 1 shows that the asymptotic complexity of IsaScheme is bounded by $O(|X|^{|\mathcal{V}(t)|})$ where $|X|$ is the number of closed terms used during the instantiation of the scheme t. This is partially mediated by the lemmata discovered during the exploration of the theory. Nevertheless, this could be improved by also exploiting the intermediate lemmata needed to finish the proofs, e.g. with the lemma calculation critic used in rippling [30].

Another limitation is that termination (and thus confluence) of rewrite systems is in general undecidable and requires sophisticated technology to solve interesting cases. This problem is aggravated by rewrite systems with a large number of rewrite rules. In this situation, termination checking demanded by definition 8 (see page 40) would benefit from modular properties of rewrite systems such as *hierarchical termination* [22]. For instance, instead of proving termination for a big set of rewrite rules, we could try to partition the set in such a way that no two subsets make reference to the same function, and then prove termination of these subsets of rules individually.

Moreover, the completion algorithm described in section 3.4 attempts to construct a convergent rewrite system from a given set of equations. However, completion does not always succeed on a set of identities and a reduction ordering. A failure occurs when an initial identity or a normal form of a critical pair cannot be oriented by the given ordering. Completion can also fail in an infinite execution of the rules in table 3.2. These problems are addressed in a completion technique called *completion without failure* [2] which, if implemented, would subsume the AC-rewriting techniques used by IsaScheme. The main advantage of this completion technique is that it allows the use of permutative rules by employing a reduction order on terms where termination is enforced in each rewrite step.

8.3 Contributions

In chapter 1, we indicated the contributions of this work to the state of the art in the scheme-based mathematical theory-exploration. Here we restate each of them giving a brief description.

8.3.1 Automation of the Instantiation Process

We designed and implemented the IsaScheme program following a new approach to mathematical theory-exploration based on schemes. Other than IsaScheme, Theorema is the only system performing the exploration of mathematical theories based on schemes [8]. However, in the latter system, the user needs to perform all schematic substitutions manually, as Theorema does not instantiate the schemes automatically from a set of terms. In IsaScheme, this process was completely automated.

We feel that the integration of existing provers, such as Isabelle, is very important in theory-formation systems because we can reuse a number of relevant algorithms during the exploration of mathematical theories. For instance, higher-order unification and type inference are rather complex algorithms. They are already implemented in the Isabelle prover and IsaScheme exploited such algorithms to automate the instantiation process of schemes.

Another important contribution is that ensuring the soundness of definitions is left to the user in Theorema. IsaScheme was developed on top of Isabelle/HOL following the LCF tradition. By using existing definitional tools we ensured that IsaScheme is conservative by construction, and thus offered a maximum of safety from unsoundness [39].

8.3.2 Applicability and Higher-order Capabilities

An important aspect of every theory-exploration system is its applicability across different mathematical theories. The scheme-based approach provides a generic mechanism for the exploration of any mathematical theory where the symbols in the theory's signature (or the closed terms constructed from them) and the free variables within the schemes could be unified. IsaScheme has been successfully applied to different mathematical theories, such as: natural numbers, lists and binary trees.

As is the case regarding IsaCoSy, IsaScheme works with Isabelle theories where important properties of datatypes (such as induction, split rules, cases, etc.) or functions (such as well-definedness, recursive induction rules, etc.) are proved automatically following the LCF approach. This facilitates the use of IsaScheme in different theories. MATHsAiD and Theorema can also be used in new domains but all the properties proved automatically by Isabelle have to be inserted manually as axioms, rendering those systems prone to human error and thus, potentially unsound.

IsaScheme is capable of synthesising higher-order conjectures or definitions con-

taining λ-terms. Examples include the synthesis of theorems using Gödel's recursor in table D.2 or the theorems depicted in table D.8 using case statements (e.g. the term $case\ y\ of\ 0 \Rightarrow suc\ x \mid suc\ z \Rightarrow suc\ (max\ z\ x)$ is represented internally in Isabelle by $case\_nat\ (suc\ x)\ (\lambda z.\, suc\ (max\ z\ x))\ y$).

8.3.3 Irreducibility as a Measure of Worth

We designed and implemented a novel technique based on termination and completion of rewrite systems to show how the new definitions and the lemmata discovered during the exploration of a theory can be used, not only to help with the proof obligations during the exploration, but also to reduce redundancies inherent in most theory-formation systems. This technique allowed us to neglect most of the existing heuristics used in mathematical theory-exploration, such as the ones mentioned in [18].

IsaCoSy also uses irreducibility as a measure of worth. However, IsaCoSy's constraint algorithm is not concerned with termination or convergence of the equations it synthesises. IsaScheme, on the contrary, strives to maintain a terminating and potentially convergent rewrite system which is used to obtain new irreducible conjectures and definitions.

8.4 Summary

We have implemented the proposed scheme-based approach to mathematical theory-exploration in Isabelle/HOL for the generation of conjectures and definitions, and have shown how the instantiation process of schemes can be automated. We have also described how we can make productive use of normalisation in two ways: first to improve proof automation by maintaining a terminating and potentially convergent rewrite system, and second to avoid numerous redundancies inherent in most theory-exploration systems.

We performed a precision/recall analysis with Isabelle's libraries to evaluate the quality of the theorems and definitions found by our method and implementation. IsaScheme produces many interesting theorems resulting in high precision of 100% for the natural numbers and 70% for the theory of lists. IsaScheme finds all the theorems for the theory of lists resulting in a recall of 100% but it does not find all the theorems for the naturals, obtaining only 46%. Surprisingly, all theorems about the naturals not found were subsumed by the theorems IsaScheme synthesised. IsaScheme produced

all the definitions for the theory of naturals after 3 exploration rounds producing a recall of 100%. However, IsaScheme produced a low recall of definitions for the theory of lists obtaining only 21%. The culprit was the definitional scheme used, as it was only able to synthesize binary functions and not unary or ternary ones. IsaScheme produced a low precision for definitions in both theories, obtaining 8% for the natural numbers and 25% for the theory of lists.

Appendix A

Datatypes and Definition of Functions

A.1 Natural Numbers

datatype $nat \;=\; 0 \mid suc\ nat$

Addition $(+ :: nat \to nat \to nat)$

$$\begin{aligned} 0+y &= y \\ (suc\ x)+y &= suc\ (x+y) \end{aligned}$$

Multiplication $(* :: nat \to nat \to nat)$

$$\begin{aligned} 0*y &= 0 \\ (suc\ x)*y &= y+(x*y) \end{aligned}$$

Exponentiation $(\hat{} :: nat \to nat \to nat)$

$$\begin{aligned} 0\hat{}y &= suc\ 0 \\ (suc\ x)\hat{}y &= y*(x\hat{}y) \end{aligned}$$

Minus $(- :: nat \to nat \to nat)$

$$\begin{aligned} 0-y &= 0 \\ (suc\ x)\ -\ y &= case\ y\ of\ 0 \to suc\ x \mid suc\ z \to x\ -\ z \end{aligned}$$

Less $(<:: nat \to nat \to bool)$

$$\begin{aligned} x<0 &= False \\ x<(suc\ y) &= case\ x\ of\ 0 \to True \mid suc\ z \to z<y \end{aligned}$$

Greater $(>:: nat \to nat \to bool)$

$$\begin{aligned} 0>y &= False \\ (suc\ x)>y &= case\ y\ of\ 0 \to True \mid suc\ z \to x>z \end{aligned}$$

Less_eq ($\leq :: nat \to nat \to bool$)

$$\begin{aligned} 0 \leq y &= True \\ (suc\ x) \leq y &= case\ y\ of\ 0 \to False \mid suc\ z \to x \leq z \end{aligned}$$

Max ($max :: nat \to nat \to nat$)

$$\begin{aligned} max\ 0\ y &= y \\ max\ (suc\ x)\ y &= case\ y\ of\ 0 \to suc(x) \mid suc\ z \to suc\ (max\ x\ z) \end{aligned}$$

Min ($min :: nat \to nat \to nat$)

$$\begin{aligned} min\ 0\ y &= 0 \\ min\ (suc\ x)\ y &= case\ y\ of\ 0 \to 0 \mid suc\ z \to suc\ (min\ x\ z) \end{aligned}$$

Even ($even :: nat \to bool$)

$$\begin{aligned} even\ 0 &= True \\ even\ (suc\ 0) &= False \\ even\ (suc\ (suc\ x)) &= even\ x \end{aligned}$$

Odd ($odd :: nat \to bool$)

$$\begin{aligned} odd\ 0 &= False \\ odd\ (suc\ 0) &= True \\ odd\ (suc\ (suc\ x)) &= odd\ x \end{aligned}$$

A.2 Lists

datatype $'a\ list = [] \mid \#\ of\ suc\ nat$

Append ($@ :: `a\ list \to `a\ list \to `a\ list$)

$$\begin{aligned} [] @ l &= l \\ (h \# t) @ l &= h \# (t @ l) \end{aligned}$$

List reverse ($rev :: `a\ list \to `a\ list$)

$$\begin{aligned} rev\ [] &= [] \\ rev\ (h \# t) &= (rev\ t) @ (h \# []) \end{aligned}$$

Tail recursive reverse ($qrev :: `a\ list \to `a\ list \to `a\ list$)

$$\begin{aligned} qrev\ []\ y &= y \\ qrev\ (z \# x)\ y &= qrev\ x\ (z \# y) \end{aligned}$$

Map ($map :: (`a \to `b) \to `a\ list \to `b\ list$)

$$\begin{aligned} map\ f\ [] &= [] \\ map\ f\ (a \# b) &= (f\ a) \# (map\ f\ b) \end{aligned}$$

Maps ($maps :: (`a \to `b\ list) \to `a\ list \to `b\ list$)

$$\begin{aligned} maps\ f\ [] &= [] \\ maps\ f (a \# b) &= (f\ a)\ @\ (maps\ f\ b) \end{aligned}$$

Length ($len :: `a\ list \to nat$)

$$\begin{aligned} len\ [] &= 0 \\ len\ (h \# t) &= suc\ (len\ t) \end{aligned}$$

Member ($member :: `a \to `a\ list \to bool$)

$$\begin{aligned} member\ x\ [] &= False \\ member\ x\ (h \# t) &= if\ x = h\ then\ True\ else\ member\ x\ t \end{aligned}$$

Count ($count :: `a \to `a\ list \to nat$)

$$\begin{aligned} count\ x\ [] &= 0 \\ count\ x\ (h \# t) &= if\ x = h\ then\ (suc\ 0) + (count\ x\ t)\ else\ count\ x\ t \end{aligned}$$

Concat ($concat :: `a\ list\ list \to `a\ list$)

$$\begin{aligned} concat\ [] &= [] \\ concat\ (a \# b) &= a\ @\ concat\ b \end{aligned}$$

Zip ($zip :: `a\ list \to `b\ list \to (`a * `b)\ list$)

$$\begin{aligned} zip\ xs\ [] &= [] \\ zip\ xs\ (y \# ys) &= case\ xs\ of\ [] \to [] \mid (z \# zs) \to (z,y) \# zip\ zs\ ys \end{aligned}$$

Filter ($filter :: (`a \to bool) \to `a\ list \to `a\ list$)

$$\begin{aligned} filter\ p\ [] &= [] \\ filter\ p\ (a \# b) &= if\ p\ a\ then\ a \#\ filter\ p\ b\ else\ filter\ p\ b \end{aligned}$$

Insert ($insert :: nat \to nat\ list \to nat\ list$)

$$\begin{aligned} insert\ x\ [] &= x \# [] \\ insert\ x\ (a \# b) &= if\ x < a\ then\ x \# a \# b\ else\ insert\ x\ b \end{aligned}$$

Insert_1 ($insert\_1 :: nat \to nat\ list \to nat\ list$)

$$\begin{aligned} insert\_1\ x\ [] &= x\ \#\ [] \\ insert\_1\ x\ (a\ \#\ b) &= if\ x = a\ then\ x\ \#\ b\ else\ a\ \#\ insert\_1\ x\ b \end{aligned}$$

Delete ($delete :: nat \to nat\ list \to nat\ list$)

$$\begin{aligned} delete\ x\ [] &= x\ \#\ [] \\ delete\ x\ (a\ \#\ b) &= if\ x = a\ then\ delete\ x\ b\ else\ a\ \#\ delete\ x\ b \end{aligned}$$

Sort ($sort :: nat\ list \to nat\ list$)

$$\begin{aligned} sort\ [] &= [] \\ sort\ (a\ \#\ b) &= insert\ a\ (sort\ b) \end{aligned}$$

Sorted ($sorted :: nat\ list \to bool$)

$$\begin{aligned} sorted\ [] &= True \\ sorted\ (h\ \#\ t) &= case\ t\ of\ [] \to True \mid (h2\ \#\ t2) \to (h \leq h2) \wedge sorted(h2\ \#\ t2) \end{aligned}$$

Fold-left ($foldl :: (`a \to `b \to `a) \to `a \to `b\ list \to `a$)

$$\begin{aligned} foldl\ f\ a\ [] &= a \\ foldl\ f\ a\ (x\ \#\ xs) &= foldl\ f\ (f\ a\ x)\ xs \end{aligned}$$

Fold-right ($foldr :: (`a \to `b \to `b) \to `a\ list \to `b \to `b$)

$$\begin{aligned} foldr\ f\ []\ a &= a \\ foldr\ f\ (x\ \#\ xs)a &= f\ x\ (foldl\ f\ xs\ a) \end{aligned}$$

Binary head ($hd :: `a\ list \to `a \to `a$)

$$\begin{aligned} hd\ []\ y &= y \\ hd\ (z\ \#\ x)\ y &= z \end{aligned}$$

last ($last :: `a\ list \to `a \to `a$)

$$last\ (h\ \#\ t) = if\ t = []\ then\ h\ else\ last\ t$$

Binary last ($blast :: `a\ list \to `a \to `a$)

$$\begin{aligned} blast\ []\ y &= y \\ blast\ (z\ \#\ x)\ y &= last\ x\ z \end{aligned}$$

Butlast ($butlast :: `a\ list \to `a\ list$)

$$\begin{aligned} blast\ [] &= [] \\ blast\ (h\ \#\ t) &= if\ t = []\ then\ []\ else\ h\ \#\ butlast\ t \end{aligned}$$

Drop ($drop :: nat \to \text{`}a\ list \to \text{`}a\ list$)

$$\begin{aligned} drop\ n\ [] &= [] \\ drop\ n\ (h \# t) &= case\ n\ of\ 0 \to h \# t \mid suc\ m \to drop\ m\ t \end{aligned}$$

DropWhile ($dropWhile :: (\text{`}a \to bool) \to \text{`}a\ list \to \text{`}a\ list$)

$$\begin{aligned} dropWhile\ p\ [] &= [] \\ dropWhile\ p\ (h \# t) &= if\ p\ h\ then\ dropWhile\ p\ t\ else\ h \# t \end{aligned}$$

Take ($take :: nat \to \text{`}a\ list \to \text{`}a\ list$)

$$\begin{aligned} take\ n\ [] &= [] \\ take\ n\ (h \# t) &= case\ n\ of\ 0 \to [] \mid suc\ m \to h \# take\ m\ t \end{aligned}$$

TakeWhile ($takeWhile :: (\text{`}a \to bool) \to \text{`}a\ list \to \text{`}a\ list$)

$$\begin{aligned} takeWhile\ p\ [] &= [] \\ takeWhile\ p\ (h \# t) &= if\ p\ h\ then\ h \# takeWhile\ p\ t\ else\ [] \end{aligned}$$

Replicate ($replicate :: \text{`}a\ list \to \text{`}b \to \text{`}b\ list$)

$$\begin{aligned} replicate\ []\ y &= y \\ replicate\ (z \# x)\ y &= y \# replicate\ x\ y \end{aligned}$$

A.3 Trees

datatype $'a\ tree = Leaf \mid Node\ (\text{`}a\ tree)\ \text{`}a\ (\text{`}a\ tree)$

Mirror ($mirror :: \text{`}a\ tree \to \text{`}a\ tree$)

$$\begin{aligned} mirror\ Leaf &= Leaf \\ mirror\ (Node\ l\ data\ r) &= Node\ (mirror\ r)\ data\ (mirror\ l) \end{aligned}$$

Nodes ($nodes :: \text{`}a\ tree \to nat$)

$$\begin{aligned} nodes\ Leaf &= 0 \\ nodes\ (Node\ l\ data\ r) &= (suc\ 0) + (nodes\ l) + (nodes\ r) \end{aligned}$$

Height ($height :: \text{`}a\ tree \to nat$)

$$\begin{aligned} height\ Leaf &= 0 \\ height\ (Node\ l\ data\ r) &= suc\ (max\ (height\ l)\ (height\ r)) \end{aligned}$$

Appendix B

Example Sessions

We present two sessions using IsaScheme with the naturals. Due to space considerations we have chosen to illustrate simple and small examples highlighting most of IsaScheme's functionality, rather than complicated and longer ones. The first session performs an exploration with the algorithm `InventTheorems` and considers the addition of natural numbers represented with Gödel's recursor by the term $\lambda x\,y.\ rec\ x\ y\ (\lambda u\,v.\ suc\ v)$. The second session performs an exploration with the algorithm `InventDefinitions` on a theory with the datatype of naturals.

B.1 Theorem-synthesis Example Session

Below we describe an execution trace of `InventTheorems` with a theory of naturals and addition defined on top of Gödel's recursor. As a consequence of the high number of synthesised conjectures (6244 conjectures in this case), we only report unfalsified conjectures.

Example 13. *Let $\mathcal{F}$ be a signature consisting of $\mathcal{F} := \{suc : nat \to nat, 0 : nat, \oplus : nat \to nat \to nat, rec : nat \to \tau \to (nat \to \tau \to \tau) \to \tau\}$. Since the InventTheorems receives a proof technique $\mathcal{P}$ as a parameter, we assume an omniscient proof procedure which finds a proof if it exists. Also let s_1 and s_2 be the following schemes (here we assume the most general types inferred for the schemes).*

$$s_1 := \left(\begin{array}{l} assoc\text{-}scheme\ P \equiv \\ \qquad \forall x\,y\,z.\ P\ (P\ x\ y)\ z = P\ x\ (P\ y\ z) \end{array} \right)$$

$$s_2 := \left(\begin{array}{l} scheme\text{-}binary\ P\ Q\ R\ S\ T\ U \equiv \\ \qquad \forall x\,y\,z.\ P\ (Q\ x\ y)\ (R\ x\ z) = S\ (T\ x\ z)\ (U\ y\ z) \end{array} \right)$$

and assuming the following closed terms X

$$X := \left\{ \begin{array}{c} (\lambda xy.\ x), (\lambda xy.\ y), (\lambda x.\ suc), (\lambda xy.\ suc\ x), \\ (\lambda xy.\ 0), (\lambda xy.\ suc\ 0), \oplus \end{array} \right\}.$$

Moreover, let $\mathcal{R}$ be the convergent rewrite system

$$\mathcal{R} := \left\{ \begin{array}{rcl} rec\ 0\ y\ z & \to & y \\ rec\ (suc\ x)\ y\ z & \to & z\ x\ (rec\ x\ y\ z) \\ \oplus & \to & (\lambda xy.\ rec\ x\ y\ (\lambda uv.\ suc\ v)) \end{array} \right\}.$$

The first unfalsified conjecture is obtained with the substitution $\sigma_1 := \{P \mapsto \oplus\}$ on s_1. σ_1 produces the normal form of the associativity theorem for addition with the instantiation $inst(s_1, \sigma_1) := \forall xyz.\ rec\ (rec\ x\ y\ (\lambda u.\ suc))\ z\ (\lambda u.\ suc) = rec\ x\ (rec\ y\ z\ (\lambda u.\ suc))\ (\lambda u.\ suc)$ which is proved by $\mathcal{P}$. The algorithm `InventTheorems` *now constructs the normalizing extension of $\mathcal{R}$ and $inst(s_1, \sigma_1)$ producing a new $\mathcal{R}$ after a successful execution of completion*

$$\mathcal{R} := \left\{ \begin{array}{rcl} rec\ 0\ y\ z & \to & y \\ rec\ (suc\ x)\ y\ z & \to & z\ x\ (rec\ x\ y\ z) \\ \oplus & \to & (\lambda xy.\ rec\ x\ y\ (\lambda uv.\ suc\ v)) \\ rec\ (rec\ x\ y\ (\lambda u.\ suc))\ z\ (\lambda u.\ suc) & \to & rec\ x\ (rec\ y\ z\ (\lambda u.\ suc))\ (\lambda u.\ suc) \end{array} \right\}.$$

The second unfalsified conjecture is obtained with the substitution $\sigma_2 := \{P \mapsto \oplus, Q \mapsto (\lambda xy.x), R \mapsto (\lambda xy.0), S \mapsto (\lambda xy.x), T \mapsto (\lambda xy.x)\}$ on s_2. This substitution produces the instantiation $inst(s_2, \sigma_2) := \forall x.rec(x, 0, (\lambda u.suc)) = x$ which is proved by $\mathcal{P}$. The normalizing extension of $\mathcal{R}$ and $\forall x.rec(x, 0, (\lambda u.suc)) = x$ produces a new $\mathcal{R}$ after a successful execution of completion

$$\mathcal{R} := \left\{ \begin{array}{rcl} rec\ 0\ y\ z & \to & y \\ rec\ (suc\ x)\ y\ z & \to & z\ x\ (rec\ x\ y\ z) \\ \oplus & \to & (\lambda xy.\ rec\ x\ y\ (\lambda uv.\ suc\ v)) \\ rec\ (rec\ x\ y\ (\lambda u.\ suc))\ z\ (\lambda u.\ suc) & \to & rec\ x\ (rec\ y\ z\ (\lambda u.\ suc))\ (\lambda u.\ suc) \\ rec\ x\ 0\ (\lambda u.\ suc) & \to & x \end{array} \right\}.$$

The third substitution $\sigma_3 := \{P \mapsto \oplus, Q \mapsto (\lambda xy.\ x), R \mapsto (\lambda xy.\ suc\ 0), S \mapsto (\lambda xy.\ x), T \mapsto (\lambda xy.\ suc\ x)\}$ on s_2 produces the instantiation $inst(s_2, \sigma_3) := \forall x.\ rec\ x\ (suc\ 0)\ (\lambda u.\ suc) = suc\ x$ which is proved by $\mathcal{P}$. The normalizing extension of $\mathcal{R}$ and $inst(s_2, \sigma_3)$ produces

a new $\mathcal{R}$ after a successful execution of completion

$$\mathcal{R} := \left\{ \begin{array}{rcl} rec\ 0\ y\ z & \to & y \\ rec\ (suc\ x)\ y\ z & \to & z\ x\ (rec\ x\ y\ z) \\ \oplus & \to & (\lambda xy.\ rec\ x\ y\ (\lambda uv.\ suc\ v)) \\ rec\ (rec\ x\ y\ (\lambda u.\ suc))\ z\ (\lambda u.\ suc) & \to & rec\ x\ (rec\ y\ z\ (\lambda u.\ suc))\ (\lambda u.\ suc) \\ rec\ x\ 0\ (\lambda u.\ suc) & \to & x \\ rec\ x\ (suc\ y)\ (\lambda u.\ suc) & \to & suc\ (rec\ x\ y\ (\lambda u.\ suc)) \end{array} \right\}.$$

Note that the theorem $rec\ x\ (suc\ 0)\ (\lambda u.\ suc) = suc\ x$ is discarded by completion (as in example 12) and in turn produces a more general version of the theorem automatically, i.e. $rec\ x\ (suc\ y)\ (\lambda u.\ suc) = suc\ (rec\ x\ y\ (\lambda u.\ suc))$.

The fourth substitution $\sigma_4 := \{P \mapsto \oplus, Q \mapsto (\lambda xy.\ y), R \mapsto (\lambda xy.\ x), S \mapsto +, T \mapsto (\lambda xy.\ x), U \mapsto (\lambda xy.\ x)\}$ on s_2 produces the instantiation $inst(s_2, \sigma_4) := \forall xy.\ rec\ x\ y\ (\lambda u.\ suc) = rec\ y\ x\ (\lambda u.\ suc)$ (again proved by $\mathcal{P}$). This instantiation invents the normal form of commutativity of $\oplus$ which is required by AC-rewriting. However, this equation cannot be oriented by the techniques described in section 3.3 and also cannot be used as an ordered rewriting rule because IsaScheme has not found the equivalent equation (4.11) for $\oplus$ also required by AC-rewriting. Thus, completion and termination fail leaving $\mathcal{R}$ unchanged.

The fifth substitution $\sigma_5 := \{P \mapsto \oplus, Q \mapsto \oplus, R \mapsto (\lambda xy.\ y), S \mapsto \oplus, T \mapsto \oplus, U \mapsto (\lambda xy.\ x)\}$ on s_2 produces the instantiation $inst(s_2, \sigma_5) := \forall xyz.\ rec\ x\ (rec\ y\ z\ (\lambda u.\ suc))\ (\lambda u.\ suc) = rec\ x\ (rec\ z\ y\ (\lambda u.suc))\ (\lambda u.\ suc)$. This instantiation is another permutative rule such as commutativity of $\oplus$, previously discovered, and cannot be oriented by the techniques described in section 3.3. Hence, completion and termination fail again leaving $\mathcal{R}$ unchanged.

The sixth substitution $\sigma_6 := \{P \mapsto \oplus, Q \mapsto (\lambda xy.\ y), R \mapsto \oplus, S \mapsto \oplus, T \mapsto \oplus, U \mapsto (\lambda xy.\ x)\}$ on s_2 produces the instantiation $inst(s_2, \sigma_6) := \forall xyz.\ rec\ x\ (rec\ z\ y\ (\lambda u.\ suc))\ (\lambda u.suc) = rec\ y\ (rec\ x\ z\ (\lambda u.\ suc))\ (\lambda u.\ suc)$. This instantiation again is a permutative consequence of associativity and commutativity of $\oplus$ and cannot be oriented by the techniques described in section 3.3. Hence, completion and termination fail again leaving $\mathcal{R}$ unchanged.

The seventh substitution $\sigma_7 := \{P \mapsto \oplus, Q \mapsto (\lambda xy.\ y), R \mapsto \oplus, S \mapsto \oplus, T \mapsto (\lambda xy.\ x), U \mapsto \oplus\}$ on s_2 produces the instantiation $inst(s_2, \sigma_7) := \forall xyz.\ rec\ x\ (rec\ y\ z\ (\lambda u.\ suc))\ (\lambda u.\ suc) = rec\ y\ (rec\ x\ z\ (\lambda u.\ suc))\ (\lambda u.\ suc)$. This is the last equation required by AC-rewriting and thus IsaScheme turns the relevant equations into ordered rewrite rules producing

a new $\mathcal{R}$

$$\mathcal{R} := \left\{ \begin{array}{rcl} rec\ 0\ y\ z & \rightarrow & y \\ rec\ (suc\ x)\ y\ z & \rightarrow & z\ x\ (rec\ x\ y\ z) \\ \oplus & \rightarrow & (\lambda xy.\ rec\ x\ y\ (\lambda uv.\ suc\ v)) \\ rec\ (rec\ x\ y\ (\lambda u.\ suc))\ z\ (\lambda u.\ suc) & \rightarrow & rec\ x\ (rec\ y\ z\ (\lambda u.\ suc))\ (\lambda u.\ suc) \\ rec\ x\ 0\ (\lambda u.\ suc) & \rightarrow & x \\ rec\ x\ (suc\ y)\ (\lambda u.\ suc) & \rightarrow & suc\ (rec\ x\ y\ (\lambda u.\ suc)) \\ rec\ x\ y\ (\lambda u.\ suc) & \leftrightarrow & rec\ y\ x\ (\lambda u.\ suc) \\ rec\ x\ (rec\ y\ z\ (\lambda u.\ suc))\ (\lambda u.\ suc) & \leftrightarrow & rec\ y\ (rec\ x\ z\ (\lambda u.\ suc))\ (\lambda u.\ suc) \end{array} \right\}.$$

Note that the theorems produced by the instantiations σ_5 *and* σ_6 *are now normalised to True (w.r.t.* $\mathcal{R}$ *and the rules in table 4.5) and thus are discarded by IsaScheme. The* `InventTheorems` *algorithm finishes its execution producing a total of 6244 instantiations. There are 287 falsified conjectures and a total of 5702 equivalent instantiations. Only 248 instantiations were subsumed by the theorems synthesised. The set* T *of new theorems discovered by* `InventTheorems` *is:*

$$T := \left\{ \begin{array}{rcl} rec\ (rec\ x\ y\ (\lambda u.\ suc))\ z\ (\lambda u.\ suc) & \rightarrow & rec\ x\ (rec\ y\ z\ (\lambda u.\ suc))\ (\lambda u.\ suc) \\ rec\ x\ 0\ (\lambda u.\ suc) & \rightarrow & x \\ rec\ x\ (suc\ y)\ (\lambda u.\ suc) & \rightarrow & suc\ (rec\ x\ y\ (\lambda u.\ suc)) \\ rec\ x\ y\ (\lambda u.\ suc) & \leftrightarrow & rec\ y\ x\ (\lambda u.\ suc) \\ rec\ x\ (rec\ y\ z\ (\lambda u.\ suc))\ (\lambda u.\ suc) & \leftrightarrow & rec\ y\ (rec\ x\ z\ (\lambda u.\ suc))\ (\lambda u.\ suc) \end{array} \right\}.$$

B.2 Definition-synthesis Example Session

Below we describe an execution trace of the algorithm `InventDefinitions` with a theory of naturals.

Example 14. *We continue example 12 by including the definitional scheme*

$$\left(\begin{array}{l} def\text{-}scheme(G,H,I,J) \equiv \\ \qquad \exists f.\ \forall xy.\quad \bigwedge \left\{ \begin{array}{l} f(G,y) = H(y) \\ f(I(x),y) = J(y,f(x,y)) \end{array} \right. \end{array} \right).$$

Assume that the set of closed terms used to instantiate the aforementioned definitional scheme is

$$X_d := \left\{ \begin{array}{c} (\lambda x.x), (\lambda xy.x), (\lambda xy.y), \\ (\lambda x.0), (\lambda x.Suc(0)), + \end{array} \right\}.$$

The first instantiation of the definitional scheme is obtained with the substitution $\sigma_1 := \{G \mapsto 0, H \mapsto (\lambda x.0), I \mapsto suc, J \mapsto (\lambda xy.x)\}$ *producing the definition* $f_1 : nat \to nat \to nat$

$$\begin{aligned} f_1(0,y) &= 0 \\ f_1(suc(x),y) &= y. \end{aligned}$$

The recursive call to the `InventTheorems` *algorithm eventually generates the 11 theorems*

$$\begin{array}{ccc} f_1(x,f_1(x,z)) = f_1(x,z) & f_1(x,y+z) = f_1(x,y) + f_1(x,z) & f_1(x,0) = 0 \\ f_1(z,f_1(y,z)) = f_1(y,z) & f_1(x,f_1(y,z)) = f_1(y,f_1(x,z)) & f_1(x,x) = x \\ f_1(x+y,x) = x & f_1(f_1(x,y),z) = f_1(x,f_1(y,z)) & f_1(x+z,z) = z \\ f_1(x+y,f_1(x,z)) = f_1(x,z) & f_1(x+z,f_1(y,z)) = f_1(y,z). & \end{array}$$

The second instantiation of the definitional scheme is obtained with the substitution $\sigma_2 := \{G \mapsto 0, H \mapsto (\lambda x.0), I \mapsto suc, J \mapsto (\lambda x.x)\}$ *producing the definition* $f_2 : nat \to (nat \to nat) \to nat$

$$\begin{aligned} f_2(0,y) &= 0 \\ f_2(suc(x),y) &= y(f_2(x,y)). \end{aligned}$$

The recursive call to the `InventTheorems` *algorithm produces no theorems this time. The third instantiation of the definitional scheme is obtained with the substitution* $\sigma_3 := \{G \mapsto 0, H \mapsto (\lambda x.0), I \mapsto suc, J \mapsto +\}$ *producing the definition of multiplication* $f_3 : nat \to nat \to nat$

$$\begin{aligned} f_3(0,y) &= 0 \\ f_3(suc(x),y) &= y + f_3(x,y). \end{aligned}$$

The recursive call to the `InventTheorems` *algorithm generates the seven theorems*

$$\begin{array}{cc} f_3(x,0) = 0 & f_3(x,y) = f_3(y,x) \\ f_3(x,suc(z)) = x + f_3(x,z) & f_3(x,f_3(y,z)) = f_3(y,f_3(x,z)) \\ f_3(x,y+z) = f_3(x,y) + f_3(x,z) & f_3(f_3(x,y),z) = f_3(x,f_3(y,z)) \\ f_3(x+y,z) = f_3(x,z) + f_3(y,z). & \end{array}$$

The fourth instantiation of the definitional scheme is obtained with the substitution $\sigma_4 := \{G \mapsto 0, H \mapsto (\lambda x.suc(0)), I \mapsto suc, J \mapsto (\lambda xy.x)\}$ *producing the definition* $f_4 : nat \to nat \to nat$

$$\begin{aligned} f_4(0,y) &= suc(0) \\ f_4(suc(x),y) &= y. \end{aligned}$$

The recursive call to the `InventTheorems` *algorithm generates the nine theorems*

$$\begin{array}{cc} f_4(x,suc(0)) = suc(0) & f_4(x,suc(x)) = suc(x) \\ f_4(x,\, f_4(x,z)) = f_4(x,z) & f_4(x,\, f_4(y,z)) = f_4(y,\, f_4(x,z)) \\ f_4(\, f_4(x,z), suc(z)) = suc(z) & f_4(\, f_4(x,y)),\, f_4(x,z)) = f_4(x,\, f_4(y,z)) \\ f_4(x+y, suc(x)) = suc(x) & f_4(x+z, suc(z)) = suc(z) \\ f_4(x+y,\, f_4(x,z)) = f_4(x,z). & \end{array}$$

The substitution $\sigma_4 := \{G \mapsto 0, H \mapsto (\lambda x.\, suc(0)), I \mapsto suc, J \mapsto (\lambda xy.\, x)\}$ *generates the definition* $f_5 : nat \to (nat \to nat) \to nat$

$$\begin{array}{rcl} f_5(0,y) & = & suc(0) \\ f_5(suc(x),y) & = & y(\, f_5(x,y)). \end{array}$$

The recursive call to the `InventTheorems` *algorithm does not generate any theorems. The sixth instantiation of the definitional scheme is obtained with the substitution* $\sigma_6 := \{G \mapsto 0, H \mapsto (\lambda x.\, suc(0)), I \mapsto suc, J \mapsto +\}$ *producing the definition* $f_6 : nat \to nat \to nat$

$$\begin{array}{rcl} f_6(0,y) & = & suc(0) \\ f_6(suc(x),y) & = & y + f_6(x,y). \end{array}$$

The recursive call to the `InventTheorems` *algorithm generates the three theorems*

$$f_6(x,0) = suc(0) \quad f_6(x,y) = f_6(y,x) \quad f_6(x,suc(z)) = x + f_6(x,z).$$

The seventh instantiation of the definitional scheme is obtained with the substitution $\sigma_7 := \{G \mapsto 0, H \mapsto suc, I \mapsto suc, J \mapsto (\lambda xy.\, x)\}$ *producing the definition* $f_7 : nat \to nat \to nat$

$$\begin{array}{rcl} f_7(0,y) & = & suc(y) \\ f_7(suc(x),y) & = & y. \end{array}$$

The recursive call to the `InventTheorems` *algorithm generates the two theorems*

$$f_7(y,suc(z)) = suc(\, f_7(y,z)) \quad f_7(x,F_7(y,z)) = f_7(y,\, f_7(x,z)).$$

The eighth instantiation of the definitional scheme is obtained with the substitution $\sigma_8 := \{G \mapsto 0, H \mapsto suc, I \mapsto suc, J \mapsto +\}$ *producing the definition* $f_8 : nat \to nat \to nat$

$$\begin{array}{rcl} f_8(0,y) & = & suc(y) \\ f_8(suc(x),y) & = & y + f_8(x,y). \end{array}$$

The recursive call to the `InventTheorems` *algorithm generates the two theorems*

$$f_8(x,0) = suc(0) \quad f_8(x,suc(z)) = suc(x + f_8(x,z)).$$

The ninth instantiation of the definitional scheme is obtained with the substitution $\sigma_9 := \{G \mapsto 0, H \mapsto (\lambda x.x), I \mapsto suc, J \mapsto (\lambda xy.x)\}$ *producing the argument neglecting definition* $f_9 : nat \to nat \to nat$

$$\begin{aligned} f_9(0,y) &= y \\ f_9(suc(x),y) &= y. \end{aligned}$$

The InventDefinitions algorithm now tests if the definition is argument neglecting and proves the required theorem $f_9(x,y) = y$ *by induction, and thus, this definition is rejected and not further explored. The last instantiation of the definitional scheme is obtained with the substitution* $\sigma_{10} := \{G \mapsto 0, H \mapsto (\lambda x.x), I \mapsto suc, J \mapsto +\}$ *producing the definition* $f_{10} : nat \to nat \to nat$

$$\begin{aligned} f_{10}(0,y) &= y \\ f_{10}(suc(x),y) &= y + f_{10}(x,y). \end{aligned}$$

The recursive call to the `InventTheorems` *algorithm generates the four theorems*

$$f_{10}(x,0) = 0 \qquad f_{10}(x,suc(z)) = suc(x + f_{10}(x,z))$$
$$f_{10}(x, f_{10}(y,z) = f_{10}(y, f_{10}(x,z)) \qquad f_{10}(x,y+z) = f_{10}(x,y) + f_{10}(x,z).$$

Appendix C

Experimental Comparison of Exhaustive Rewriting and Rippling

Table C.1 contains the corpus of 92 theorems used to compare the inductive tactic described in section 5.4 with IsaPlanner. The 'Time1' and 'Time2' columns indicate the time spent during proofs (in the minimal lemma configuration) by the `Induct_auto_tac` and IsaPlanner respectively. A time in bold indicates the time spent before failure and the label '-' indicates that the time out was reached before failure (or success). Theorems not proved in the maximal lemma configuration (where the theorems proved where available to the provers) are marked with '*'. The overall statistics are given in Table C.2.

No.	Theorem	Time1	Time2
1	$x + 0 = x$	0.042	0.004
2	$y + suc\ z = suc\ (y + z)$	0.086	0.009
3	$(x + y) + z = x + (y + z)$	0.229	0.012
4	$x + y = y + x$	0.940	0.342
5	$x + (y + z) = y + (x + z)$	7.238	0.012
6	$m - m = 0$	0.086	0.135
7	$n - (n + m) = 0$	0.069	0.294
8	$(n + m) - n = m$	0.128	0.300
9	$(k + m) - (k + n) = m - n$	0.339	0.140
10	$(i - j) - k = i - (j + k)$	15.947*	0.439
11	$n <= 0 = (n = 0)$	0.052	0.133
12	$n <= (n + m)$	0.036	0.265

13	$i < suc\ (i + m)$	0.046	0.399
14	$(min\ a\ b = a) = a <= b$	1.811	0.586
15	$(min\ a\ b = b) = b <= a$	-	0.604
16	$max\ a\ b = max\ b\ a$	0.521	0.431
17	$max\ a\ (max\ b\ c) = max\ (max\ a\ b)\ c$	-	1.022
18	$(max\ a\ b = a) = b <= a$	-	0.517
19	$(max\ a\ b = b) = a <= b$	1.994	0.519
20	$min\ a\ b = min\ b\ a$	0.396	0.456
21	$min\ a\ (min\ b\ c) = min\ (min\ a\ b)\ c$	-	1.049
22	$drop\ 0\ xs = xs$	0.015	0.133
23	$drop\ (suc\ n)\ (x\ \#\ xs) = drop\ n\ xs$	0.001	0.161
24	$drop\ n\ (map\ f\ xs) = map\ f\ (drop\ n\ xs)$	0.818	0.271
25	$len\ (drop\ n\ xs) = len\ xs - n$	0.576	0.581
26	$take\ 0\ xs = []$	0.006	0.111
27	$take\ (suc\ n)\ (x\ \#\ xs) = x\ \#\ take\ n\ xs$	0.001	0.140
28	$take\ n\ (map\ f\ xs) = map\ f\ (take\ n\ xs)$	0.634	0.632
29	$take\ n\ xs\ @\ drop\ n\ xs = xs$	0.836	0.476
30	$zip\ []\ ys = []$	0.006	0.146
31	$zip\ (x\ \#\ xs)\ ys =$ $(case\ ys\ of\ [] \Rightarrow [] \mid z\ \#\ zs \Rightarrow (x, z)\ \#\ zip\ xs\ zs)$	0.467	0.365
32	$zip\ (x\ \#\ xs)\ (y\ \#\ ys) = (x, y)\ \#\ zip\ xs\ ys$	0.001	0.163
33	$height\ (mirror\ t) = height\ t$	-*	0.013
34	$member\ x\ (l\ @\ (x\ \#\ []))$	0.031	0.015
35	$\neg\ member\ x\ (delete\ x\ l)$	0.094	**0.021***
36	$member\ x\ l \Longrightarrow member\ x\ (l\ @\ t)$	1.705	**0.011***
37	$member\ x\ t \Longrightarrow member\ x\ (l\ @\ t)$	0.294	**0.022***
38	$member\ x\ (insert\ x\ l)$	0.158	0.045
39	$member\ x\ (insert\_1\ x\ l)$	0.081	0.021
40	$len\ (insert\ x\ l) = suc\ (len\ l)$	0.113	0.020
41	$len\ (sort\ l) = len\ l$	-*	0.009
42	$xs = [] \Longrightarrow last\ (x\ \#\ xs) = x$	0.001	0.037
43	$suc\ 0 + count\ n\ l = count\ n\ (n\ \#\ l)$	0.001	0.107
44	$n = x \Longrightarrow suc\ 0 + count\ n\ l = count\ n\ (x\ \#\ l)$	0.001	4.974
45	$count\ n\ l + count\ n\ m = count\ n\ (l\ @\ m)$	0.682	0.050
46	$count\ n\ (x\ @\ (n\ \#\ [])) = suc\ (count\ n\ x)$	0.120	0.136

47	$count\ n\ (h\ \#\ []) + count\ n\ t = count\ n\ (h\ \#\ t)$	0.004	1.268
48	$count\ n\ l <= count\ n\ (l\ @\ m)$	0.347	**2.474**
49	$dropWhile\ (\lambda x.\ False)\ xs = xs$	0.016	0.009
50	$takeWhile\ (\lambda x.\ True)\ xs = xs$	0.014	0.009
51	$takeWhile\ P\ xs\ @\ dropWhile\ P\ xs = xs$	0.252	1.218
52	$filter\ P\ (xs\ @\ ys) = filter\ P\ xs\ @\ filter\ P\ ys$	0.641	2.423
53	$(m + n) - n = m$	-	**2.460**
54	$(suc\ m - n) - suc\ k = (m - n) - k$	2.649	**2.462**
55	$i < suc\ (m + i)$	-	**9.755**
56	$n <= (m + n)$	-	**2.443**
57	$m <= n \Longrightarrow m <= suc\ n$	**5.716**	**2.440***
58	$drop\ n\ (drop\ m\ xs) = drop\ (n + m)\ xs$	-	**6.073**
59	$drop\ n\ (xs\ @\ ys) = drop\ n\ xs\ @\ drop\ (n - len\ xs)\ ys$	-	**30.503**
60	$drop\ n\ (take\ m\ xs) = take\ (m - n)\ (drop\ n\ xs)$	-*	**8.546***
61	$drop\ n\ (zip\ xs\ ys) = zip\ (drop\ n\ xs)\ (drop\ n\ ys)$	-*	**9.758***
62	$rev\ (drop\ i\ xs) = take\ (len\ xs - i)\ (rev\ xs)$	-*	**24.388***
63	$rev\ (take\ i\ xs) = drop\ (len\ xs - i)\ (rev\ xs)$	-*	**20.857***
64	$rev\ (filter\ P\ xs) = filter\ P\ (rev\ xs)$	-*	**0.056***
65	$take\ n\ (xs\ @\ ys) = take\ n\ xs\ @\ take\ (n - len\ xs)\ ys$	-	**29.203**
66	$take\ n\ (drop\ m\ xs) = drop\ m\ (take\ (n + m)\ xs)$	-	**4.932**
67	$take\ n\ (zip\ xs\ ys) = zip\ (take\ n\ xs)\ (take\ n\ ys)$	-*	**8.667**
68	$len\ (filter\ P\ xs) <= len\ xs$	-*	**11.000***
69	$zip\ (xs\ @\ ys)\ zs =$ $zip\ xs\ (take\ (len\ xs)\ zs)\ @zip\ ys\ (drop\ (len\ xs)\ zs)$	-*	**12.187**
70	$zip\ xs\ (ys\ @\ zs) =$ $zip\ (take\ (len\ ys)\ xs)\ ys\ @zip\ (drop\ (len\ ys)\ xs)\ zs$	-*	-
71	$len\ xs = len\ ys \Longrightarrow zip\ (rev\ xs)\ (rev\ ys) = rev\ (zip\ xs\ ys)$	-*	**6.183***
72	$len\ (delete\ x\ l) <= len\ l$	-*	**11.061***
73	$x < y \Longrightarrow member\ x\ (insert\ y\ l) = member\ x\ l$	-*	**0.019***
74	$x \neq y \Longrightarrow member\ x\ (insert\ y\ l) = member\ x\ l$	-*	**0.067***
75	$sorted\ l \Longrightarrow sorted\ (insert\ x\ l)$	-*	**0.019***
76	$sorted\ (sort\ l)$	-*	**0.003***
77	$last\ (xs\ @\ (x\ \#\ [])) = x$	-*	**1.210***
78	$xs \neq [] \Longrightarrow last\ (x\ \#\ xs) = last\ xs$	0.001	**0.006**
79	$ys = [] \Longrightarrow last\ (xs\ @\ ys) = last\ xs$	-*	**0.063***

80	$ys \neq [] \Longrightarrow last\ (xs\ @\ ys) = last\ ys$	-*	**0.028***
81	$last\ (xs\ @\ ys) = (if\ ys = []\ then\ last\ xs\ else\ last\ ys)$	-*	**1.291***
82	$n < len\ xs \Longrightarrow last\ (drop\ n\ xs) = last\ xs$	-*	**1.326***
83	$butlast\ (xs\ @\ (x\ \#\ [])) = xs$	0.143	**0.025***
84	$xs \neq [] \Longrightarrow butlast\ xs\ @\ (last\ xs\ \#\ []) = xs$	0.646	**0.005***
85	$butlast\ (xs\ @\ ys) = (if\ ys = []\ then\ butlast\ xs\ else\ xs\ @\ butlast\ ys)$	-*	**2.539***
86	$butlast\ xs = take\ (len\ xs - suc\ 0)\ xs$	**4.967***	**40.825***
87	$len\ (butlast\ xs) = len\ xs - suc\ 0$	**0.386***	**17.255***
88	$len\ (delete\ x\ l) <= len\ l$	-	**12.225**
89	$count\ n\ t + count\ n\ (h\ \#\ []) = count\ n\ (h\ \#\ t)$	0.399	**21.111**
90	$count\ n\ l = count\ n\ (rev\ l)$	-*	**0.047***
91	$count\ x\ l = count\ x\ (sort\ l)$	-*	**2.474***
92	$n \neq h \Longrightarrow count\ n\ (x\ @\ (h\ \#\ [])) = count\ n\ x$	0.455	**1.338**

Table C.1: Corpus of 92 theorems and experimental results.

Statistic	Rippling		Reduction	
	Min	Max	Min	Max
Overall success (%)	52.1	70.7	56.5	69.6
Average CPU time (seg)	0.330	1.1	1.001	5.85

Table C.2: Overall statistics for rippling and the `Induct_auto_tac`.

Appendix D

Theorems Found

In each of the following tables we have three columns, indicating the theorem number, the theorem and whether this theorem is included in the rewrite system $\mathcal{R}$ constructed by IsaScheme. Note that it might not be possible to orientate some equations by the techniques described in section 3.3; in case an equation is non-orientable, it is not included in $\mathcal{R}$, and is not used during normalisation.

D.1 Natural Numbers

No.	Theorem	$\in \mathcal{R}$
1	$x+0=x$	✓
2	$x*0=0$	✓
3	$(suc\ 0)\hat{}x=suc\ 0$	✓
4	$y+suc\ z=suc\ (y+z)$	✓
5	$y+x=x+y$	✓
6	$(x+y)+z=x+(y+z)$	✓
7	$y+(x+z)=x+(y+z)$	✓
8	$(x+y)*z=x*z+y*z$	✓
9	$x*suc\ z=x+x*z$	✓
10	$x*(y+z)=x*y+x*z$	✓
11	$y*x=x*y$	✓
12	$(x*y)*z=x*(y*z)$	✓
13	$y*(x*z)=x*(y*z)$	✓
14	$(x*y)\hat{}z=x\hat{}z*y\hat{}z$	✓

15	$x\hat{}(y+z) = x\hat{}y * x\hat{}z$	✓
16	$(x\hat{}y)\hat{}z = x\hat{}(y*z)$	✓

Table D.1: Theorems found about addition, multiplication and exponentiation in the theory of natural numbers. The third column indicates whether the theorem is included in the rewrite system $\mathcal{R}$.

No.	Theorem	$\in \mathcal{R}$
1	$rec\ x\ 0\ (\lambda u\ v.\ v) = 0$	✓
2	$rec\ x\ 0\ (\lambda u.\ suc) = x$	✓
3	$rec\ x\ 0\ (\lambda u\ v.\ rec\ v\ y\ (\lambda u.\ suc)) = rec\ y\ 0\ (\lambda u\ v.\ rec\ v\ x\ (\lambda u.\ suc))$	✓
4	$rec\ x\ 0\ (\lambda u\ v.\ rec\ v\ (rec\ y\ 0\ (\lambda u\ v.\ rec\ v\ z\ (\lambda u.\ suc)))\ (\lambda u.\ suc)) =$ $rec\ y\ 0\ (\lambda u\ v.\ rec\ v\ (rec\ x\ 0\ (\lambda u\ v.\ rec\ v\ z\ (\lambda u.\ suc)))\ (\lambda u.\ suc))$	✓
5	$rec\ x\ 0\ (\lambda u\ v.\ suc\ (rec\ v\ z\ (\lambda u.\ suc))) =$ $rec\ x\ (rec\ x\ 0\ (\lambda u\ v.\ rec\ v\ z\ (\lambda u.\ suc)))\ (\lambda u.\ suc)$	✓
6	$rec\ x\ 0\ (\lambda u\ v.\ rec\ v\ (rec\ y\ z\ (\lambda u.\ suc))\ (\lambda u.\ suc)) =$ $rec\ (rec\ x\ 0\ (\lambda u\ v.\ rec\ v\ y\ (\lambda u.\ suc)))$ $(rec\ x\ 0\ (\lambda u\ v.\ rec\ v\ z\ (\lambda u.\ suc)))\ (\lambda u.\ suc)$	✓
7	$rec\ x\ y\ (\lambda u.\ suc) = rec\ y\ x\ (\lambda u.\ suc)$	✓
8	$rec\ x\ (suc\ 0)\ (\lambda u\ v.\ v) = suc\ 0$	✓
9	$rec\ y\ (suc\ 0)\ (\lambda u\ v.\ rec\ (rec\ z\ (suc\ 0)\ (\lambda u\ v.\ rec\ x\ 0$ $(\lambda u\ va.rec\ v\ va\ (\lambda u.\ suc))))\ 0\ (\lambda u\ va.\ rec\ v\ va\ (\lambda u.\ suc))) =$ $rec\ z\ (suc\ 0)\ (\lambda u\ v.\ rec\ (rec\ y\ (suc\ 0)\ (\lambda u\ v.\ rec\ x\ 0$ $(\lambda u\ va.\ rec\ v\ va\ (\lambda u.\ suc))))\ 0\ (\lambda u\ va.\ rec\ v\ va\ (\lambda u.\ suc)))$	✓
10	$rec\ y\ (suc\ z)\ (\lambda u.\ suc) = suc\ (rec\ y\ z\ (\lambda u.\ suc))$	✓
11	$rec\ x\ (rec\ y\ z\ (\lambda u.\ suc))\ (\lambda u.\ suc) = rec\ y\ (rec\ x\ z\ (\lambda u.\ suc))\ (\lambda u.\ suc)$	✓
12	$rec\ (rec\ x\ 0\ (\lambda u\ v.\ rec\ v\ y\ (\lambda u.\ suc)))\ 0\ (\lambda u\ v.\ rec\ v\ z\ (\lambda u.\ suc)) =$ $rec\ x\ 0\ (\lambda u\ v.\ rec\ v\ (rec\ y\ 0\ (\lambda u\ v.\ rec\ v\ z\ (\lambda u.\ suc)))\ (\lambda u.\ suc))$	✓
13	$rec\ (rec\ y\ 0\ (\lambda u\ v.\ rec\ v\ z\ (\lambda u.\ suc)))\ (suc 0)$ $(\lambda u\ v.\ rec\ x\ 0\ (\lambda u\ va.\ rec\ v\ va\ (\lambda u.\ suc))) =$ $rec\ z\ (suc\ 0)\ (\lambda u\ v.\ rec\ (rec\ y\ (suc\ 0)\ (\lambda u\ v.\ rec\ x\ 0$ $(\lambda u\ va.\ rec\ v\ va\ (\lambda u.\ suc))))\ 0\ (\lambda u\ va.\ rec\ v\ va\ (\lambda u.\ suc)))$	✓
14	$rec\ (rec\ x\ y\ (\lambda u.\ suc))\ 0\ (\lambda u\ v.\ rec\ v\ z\ (\lambda u.\ suc)) =$ $rec\ (rec\ x\ 0\ (\lambda u\ v.\ rec\ v\ z\ (\lambda u.\ suc)))$ $(rec\ y\ 0\ (\lambda u\ v.\ rec\ v\ z\ (\lambda u.\ suc)))\ (\lambda u.\ suc)$	✓

15	$rec\ (rec\ x\ y\ (\lambda u.\ suc))\ z\ (\lambda u.\ suc) = rec\ x\ (rec\ y\ z\ (\lambda u.\ suc))\ (\lambda u.\ suc)$	✓
16	$rec\ (rec\ y\ z\ (\lambda u.\ suc))\ (suc\ 0)$ $(\lambda u\ v.\ rec\ x\ 0\ (\lambda u\ va.\ rec\ v\ va\ (\lambda u.\ suc))) =$ $rec\ (rec\ y\ (suc\ 0)\ (\lambda u\ v.\ rec\ x\ 0\ (\lambda u\ va.\ rec\ v\ va\ (\lambda u.\ suc))))\ 0$ $(\lambda u\ v.\ rec\ v\ (rec\ z\ (suc\ 0)\ (\lambda u\ v.\ rec\ x\ 0$ $(\lambda u\ va.\ rec\ v\ va\ (\lambda u.\ suc))))\ (\lambda u.\ suc))$	✓
17	$rec\ (rec\ z\ (suc\ 0)\ (\lambda u\ v.\ rec\ x\ 0\ (\lambda u\ va.\ rec\ v\ va\ (\lambda u.\ suc))))\ 0$ $(\lambda u\ v.\ rec\ v\ (rec\ z\ (suc\ 0)\ (\lambda u\ v.\ rec\ y\ 0$ $(\lambda u\ va.\ rec\ v\ va\ (\lambda u.\ suc))))\ (\lambda u.\ suc)) =$ $rec\ z\ (suc\ 0)\ (\lambda u\ v.\ rec\ (rec\ x\ 0\ (\lambda u\ v.\ rec\ v\ y\ (\lambda u.\ suc)))\ 0$ $(\lambda u\ va.\ rec\ v\ va\ (\lambda u.\ suc)))$	✓

Table D.2: Theorems found about addition, multiplication and exponentiation in the theory of natural numbers using Gödel's recursor. The third column indicates whether the theorem is included in the rewrite system $\mathcal{R}$.

D.2 Set Operators

No.	Theorem	$\in \mathcal{R}$
1	$x \cup x = x$	✓
2	$x \cup y = y \cup x$	✓
3	$x \cup (x \cup z) = x \cup z$	✓
4	$x \cup (y \cup z) = y \cup (x \cup z)$	✓
5	$x \cup (z \cup (y \lhd z)) = x \cup z$	✓
6	$x \cup (z \cup (y \trianglelefteq z)) = x \cup z$	✓
7	$x \cup y \cup z = x \cup (y \cup z)$	✓
8	$(x \rhd y) \cup x = x$	✓
9	$(x \rhd y) \cup (x \cup z) = x \cup z$	✓
10	$((x \rhd z) \rhd y) \cup ((x \rhd z) \rhd z) = ((x \rhd z) \rhd y)$	✓
11	$(x \lhd z) \cup z = z$	✓
12	$(x \lhd z) \cup (y \lhd (x \lhd z)) = (x \lhd z)$	✓
13	$(x \lhd (x \trianglelefteq z)) \cup (y \lhd (x \trianglelefteq z)) = (y \lhd (x \trianglelefteq z))$	✓
14	$(x \trianglerighteq y) \cup x = x$	✓
15	$(x \trianglerighteq y) \cup (x \cup z) = x \cup z$	✓

16	$(x \lhd z) \cup z = z$	✓
17	$(x \rhd z) = (x \rhd z) \cup ((x \rhd z) \rhd y)$	✗
18	$(x \rhd y \cup z) = (x \rhd y) \cup (x \rhd z)$	✓
19	$(x \cup y \rhd z) = (x \rhd z) \cup (y \rhd z)$	✓
20	$((x \rhd z) \rhd z) = (x \rhd z)$	✓
21	$((x \rhd y) \rhd z) = ((x \rhd z) \rhd y)$	✗
22	$((x \lhd y) \rhd z) = (x \lhd (y \rhd z))$	✓
23	$((x \lhd\!\!\cdot\, y) \rhd z) = (x \lhd\!\!\cdot\, (y \rhd z))$	✓
24	$(x \lhd z) = (x \lhd z) \cup (x \lhd (y \lhd z))$	✗
25	$(x \lhd y \cup z) = (x \lhd y) \cup (x \lhd z)$	✓
26	$(x \lhd (x \lhd z)) = (x \lhd z)$	✓
27	$(x \lhd (y \lhd z)) = (y \lhd (x \lhd z))$	✗
28	$(x \cup y \lhd z) = (x \lhd z) \cup (y \lhd z)$	✓
29	$(x \rhd\!\!= z \cup (y \lhd z)) = (x \rhd\!\!= z)$	✓
30	$(x \rhd\!\!= z \cup (y \lhd\!\!\cdot\, z)) = (x \rhd\!\!= z)$	✓
31	$(x \cup y \rhd\!\!= z) = (x \rhd\!\!= z) \cup (y \rhd\!\!= z)$	✓
32	$((x \rhd y) \rhd\!\!= z) = ((x \rhd\!\!= z) \rhd y)$	✓
33	$((x \lhd y) \rhd\!\!= z) = (x \lhd (y \rhd\!\!= z))$	✓
34	$((x \rhd\!\!= y) \rhd\!\!= z) = (x \rhd\!\!= y \cup z)$	✓
35	$((x \lhd\!\!\cdot\, y) \rhd\!\!= z) = (x \lhd\!\!\cdot\, (y \rhd\!\!= z))$	✓
36	$(x \lhd\!\!\cdot\, y \cup z) = (x \lhd\!\!\cdot\, y) \cup (x \lhd\!\!\cdot\, z)$	✓
37	$(y \lhd\!\!\cdot\, (x \lhd z)) = (x \lhd (y \lhd\!\!\cdot\, z))$	✓
38	$(x \lhd\!\!\cdot\, (x \lhd\!\!\cdot\, z)) = (x \lhd\!\!\cdot\, z)$	✓
39	$(x \lhd\!\!\cdot\, (y \lhd\!\!\cdot\, z)) = (y \lhd\!\!\cdot\, (x \lhd\!\!\cdot\, z))$	✗
40	$(x \cup y \lhd\!\!\cdot\, z) = (x \lhd\!\!\cdot\, (y \lhd\!\!\cdot\, z))$	✓
41	$((x \rhd y) \lhd\!\!\cdot\, (x \lhd\!\!\cdot\, z)) = (x \lhd\!\!\cdot\, z)$	✓
42	$((x \rhd\!\!= y) \lhd\!\!\cdot\, (x \lhd\!\!\cdot\, z)) = (x \lhd\!\!\cdot\, z)$	✓

Table D.3: Theorems found about operators in set theory. The third column indicates whether the theorem is included in the rewrite system $\mathcal{R}$.

D.3 Lists

No.	Theorem	$\in \mathcal{R}$
1	$x \mathbin{@} [\,] = x$	✓

2	$len\ (rev\ x) = len\ x$	✓
3*	$len\ (map\ y\ x) = len\ x$	✓
4	$(x\ @\ y)\ @\ z = x\ @\ (y\ @\ z)$	✓
5	$rev\ (y\ @\ x) = (rev\ x)\ @\ (rev\ y)$	✓
6	$rev\ (rev\ x) = x$	✓
7	$map\ z\ (x\ @\ y) = (map\ z\ x)\ @\ (map\ z\ y)$	✓
8	$rev\ (map\ x\ z) = map\ x\ (rev\ z)$	✓
9*	$len\ (z\ @\ x) = len\ (x\ @\ z)$	✗

Table D.4: Theorems found about append (@), rev (*rev*), map (*map*) and length (*len*) in the theory of natural numbers. Theorems marked with * are not included in Isabelle's list library. The third column indicates whether the theorem is included in the rewrite system $\mathcal{R}$.

No.	Theorem	$\in \mathcal{R}$
1*	$foldr\ \#\ z\ [\,] = z$	✓
2*	$x\ @\ z = foldr\ \#\ x\ z$	✓
3	$foldl\ z\ (foldl\ z\ x\ y)\ w = foldl\ z\ x\ (foldr\ \#\ y\ w)$	✗
4	$foldr\ z\ (foldr\ \#\ x\ y)\ w = foldr\ z\ x\ (foldr\ z\ y\ w)$	✓

Table D.5: Theorems found about append (@), foldl and foldr in the theory of lists. Theorems marked with * are not included in Isabelle's list library. The third column indicates whether the theorem is included in the rewrite system $\mathcal{R}$.

No.	Theorem	$\in \mathcal{R}$
1	$rev\ (rev\ z) = z$	✓
2	$rev\ (z\ @\ x) = rev\ x\ @\ rev\ z$	✓
3	$x\ @\ [\,] = x$	✓
4	$(z\ @\ y)\ @\ x = z\ @\ (y\ @\ x)$	✓
5	$qrev\ z\ x = rev\ z\ @\ x$	✓

Table D.6: Theorems found about append (@), reverse (*rev*) and tail recursive reverse (*qrev*) in the theory of lists. The third column indicates whether the theorem is included in the rewrite system $\mathcal{R}$.

No.	Theorem	$\in \mathcal{R}$
1	$rev\ (rev\ y) = y$	✓
2	$rev\ (x\ @\ y) = rev\ y\ @\ rev\ x$	✓
3	$len\ (rev\ z) = len\ z$	✓
4	$len\ (y\ @\ z) = len\ y\ +\ len\ z$	✓
5	$y\ @\ [\,] = y$	✓
6	$(x\ @\ y)\ @\ z = x\ @\ (y\ @\ z)$	✓
7	$z + 0 = z$	✓
8	$y + z = z + y$	✓
9	$z + suc\ x = suc\ (z + x)$	✓
10	$y + (z + x) = z + (y + x)$	✓
11	$(z + y) + x = z + (y + x)$	✓
12	$([\,] = rev\ x) = ([\,] = x)$	✓
13	$(0 = len\ x) = ([\,] = x)$	✓
14	$(x = rev\ z) = (z = rev\ x)$	✗
15	$(x = x\ @\ z) = ([\,] = z)$	✓
16	$(z = x\ @\ z) = ([\,] = x)$	✓
17	$(x = x + z) = (0 = z)$	✓
18	$(z = x + z) = (0 = x)$	✓
19	$(suc\,x = x + y) = (suc\,z = y + z)$	✗
20	$(rev\ y = rev\ z) = (y = z)$	✓
21	$(x\ @\ y = x\ @\ z) = (y = z)$	✓
22	$(x\ @\ z = y\ @\ z) = (x = y)$	✓
23	$(x + y = x + z) = (y = z)$	✓
24	$(x + z = y + z) = (x = y)$	✓

Table D.7: Theorems found about function symbols $+$, $@$, len, rev and $=$ in the theory of lists (and naturals). The third column indicates whether the theorem is included in the rewrite system $\mathcal{R}$.

D.4 Trees

No.	Theorem	$\in \mathcal{R}$
1	$mirror\ (mirror\ y) = y$	✓

2	$height\ (mirror\ y) = height\ y$	✓
3	$nodes\ (mirror\ y) = nodes\ y$	✓
4	$nodes\ (Node\ l\ data\ r) = suc\ (nodes\ l + nodes\ r)$	✓
5	$max\ x\ x = x$	✓
6	$max\ x\ y = max\ y\ x$	✓
7	$max\ y\ (suc\ x) = (case\ y\ of\ 0\ =>\ suc\ x \mid suc\ z\ =>\ suc\ (max\ z\ x))$	✓
8	$max\ x\ (max\ x\ z) = max\ x\ z$	✓
9	$max\ x\ (max\ y\ z) = max\ y\ (max\ x\ z)$	✓
10	$max\ x\ (max\ y\ (x+z)) = max\ y\ (x+z)$	✓
11	$max\ x\ (x+z) = x+z$	✓
12	$max\ z\ (x+z) = x+z$	✓
13	$max\ x\ (case\ z\ of\ 0\ =>\ suc\ y \mid suc\ z\ =>\ suc\ (max\ z\ y)) =$ $(case\ max\ x\ z\ of\ 0\ =>\ suc\ y \mid suc\ z\ =>\ suc\ (max\ z\ y))$	✓
14	$max\ (max\ z\ y)\ x = max\ z\ (max\ y\ x)$	✓
15	$z+0 = z$	✓
16	$y+z = z+y$	✓
17	$z+suc\ x = suc\ (z+x)$	✓
18	$x+max\ y\ z = max\ (x+y)\ (x+z)$	✓
19	$y+(z+x) = z+(y+x)$	✓
20	$(z+y)+x = z+(y+x)$	✓
21	$(case\ x\ of\ 0\ =>\ suc\ x \mid suc\ z\ =>\ suc\ (max\ z\ x)) = suc\ x$	✓
22	$(case\ max\ x\ z\ of\ 0\ =>\ suc\ x \mid suc\ z\ =>\ suc\ (max\ z\ x)) =$ $(case\ z\ of\ 0\ =>\ suc\ x \mid suc\ z\ =>\ suc\ (max\ z\ x))$	✓
23	$(case\ max\ y\ z\ of\ 0\ =>\ suc\ z \mid suc\ za\ =>\ suc\ (max\ za\ z)) =$ $(case\ y\ of\ 0\ =>\ suc\ z \mid suc\ za\ =>\ suc\ (max\ za\ z))$	✓

Table D.8: Theorems found about functions $+$, *max*, *mirror*, *nodes* and *height* in the theory of trees. The third column indicates whether the theorem is included in the rewrite system $\mathcal{R}$.

Bibliography

[1] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[2] L. Bachmair, N. Dershowitz, and D. A. Plaisted. Completion without failure, 1989.

[3] S. Berghofer and M. Wenzel. Inductive datatypes in hol - lessons learned in formal-logic engineering. In *Theorem Proving in Higher Order Logics: TPHOLs 99, LNCS 1690*. Springer, 1999.

[4] J. Blanchette and T. Nipkow. Nitpick: A counterexample generator for higher-order logic based on a relational model finder. *TAP*, 2009.

[5] B. Buchberger. Algorithm Invention and Verification by Lazy Thinking. In D. Petcu, V. Negru, D. Zaharie, and T. Jebelean, editors, *Proceedings of SYNASC 2003, 5th International Workshop on Symbolic and Numeric Algorithms for Scientific Computing Timisoara*, pages 2–26, Timisoara, Romania, 1-4 October 2003. Copyright: Mirton Publisher.

[6] B. Buchberger. Algorithm-supported mathematical theory exploration: A personal view and strategy. In B. Buchberger and J. A. Campbell, editors, *Artificial Intelligence and Symbolic Computation, 7th International Conference, AISC 2004*, number 3249 in Lecture Notes in Computer Science, pages 236–250. Springer, 2004.

[7] B. Buchberger and A. Craciun. Algorithm synthesis by lazy thinking: Using problem schemes. In D.Petcu, V.Negru, D.Zaharie, and T.Jebelean, editors, *Proceedings of SYNASC 2004, 6th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pages 90–106, Timisoara, Romania, 2004. Mirton Publisher.

[8] B. Buchberger, A. Craciun, T. Jebelean, L. Kovács, T. Kutsia, K. Nakagawa, F. Piroi, N. Popov, J. Robu, and M. Rosenkranz. Theorema: Towards computer-aided mathematical theory exploration. *J. Applied Logic*, 4(4):470–504, 2006.

[9] A. Bundy. The use of explicit plans to guide inductive proofs. In R. Lusk and R. Overbeek, editors, *9th International Conference on Automated Deduction*, pages 111–120. Springer-Verlag, 1988. Longer version available from Edinburgh as DAI Research Paper No. 349.

[10] A. Bundy. The automation of proof by mathematical induction. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning, Volume 1*, chapter 13. Elsevier, 2001.

[11] A. Bundy, D. Basin, D. Hutter, and A. Ireland. *Rippling: Meta-level Guidance for Mathematical Reasoning*, volume 56 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2005.

[12] A. Bundy and I. Green. An experimental comparison of rippling and exhaustive rewriting. Research paper 836, Dept. of Artificial Intelligence, University of Edinburgh, December 1996.

[13] A. Church. A formulation of the simple theory of types. *Symbolic Logic*, 5(1):56–68, 1940.

[14] K. Claessen and J. Hughes. Quickcheck: a lightweight tool for random testing of Haskell programs. In *ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 268–279, New York, NY, USA, 2000. ACM.

[15] S. Colton. *Automated Theory Formation in Pure Mathematics*. PhD thesis, Division of Informatics, University of Edinburgh, 2001.

[16] S. Colton. The HR program for theorem generation. In A. Voronkov, editor, *Proceedings of the 18th International Conference on Automated Deduction*, number 2392 in Lecture Notes in Computer Science, pages 285–289. Springer, 2002.

[17] S. Colton, A. Bundy, and T. Walsh. Automatic identification of mathematical concepts. In P. Langley, editor, *Proceedings of the Seventeenth International Conference on Machine Learning*, pages 183–190, San Francisco, USA, 2000. Morgan Kaufmann.

[18] S. Colton, A. Bundy, and T. Walsh. On the notion of interestingness in automated mathematical discovery. *International Journal of Human Computer Studies*, 53(3):351–375, 2000.

[19] S. Colton and S. Muggleton. Mathematical applications of inductive logic programming. *Machine Learning*, 64(1-3):25–64, 2006.

[20] A. Craciun and M. Hodorog. Decompositions of Natural Numbers: From a Case Study in Mathematical Theory Exploration. *Symbolic and Numeric Algorithms for Scientific Computing, International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pages 41–47, 2007.

[21] N. Dershowitz. Orderings for term rewriting systems. *Theoretical Computer Science*, 17(3):279–301, 1982.

[22] N. Dershowitz. Hierachical Termination. In *CTRS '94: Proceedings of the 4th International Workshop on Conditional and Typed Rewriting Systems*, pages 89–105, London, UK, 1995. Springer-Verlag.

[23] L. Dixon. *A Proof Planning Framework for Isabelle*. PhD thesis, University of Edinburgh, 2006.

[24] L. Dixon and J. D. Fleuriot. IsaPlanner: A prototype proof planner in Isabelle. In *Proceedings of CADE'03*, volume 2741 of *LNCS*, pages 279–283, 2003.

[25] H. Ganzinger and J. Zhang. System description: MCS: Model-based Conjecture Searching. In *CADE-16: Proceedings of the 16th International Conference on Automated Deduction*, pages 393–397, London, UK, 1999. Springer-Verlag.

[26] M. J. Gordon, A. J. Milner, and C. P. Wadsworth. *Edinburgh LCF - A mechanised logic of computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

[27] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, 1993.

[28] J. Harrison. HOL light: a tutorial introduction. In M. Srivas and A. Camilleri, editors, *Proceedings of the First International Conference on Formal Methods in Computer-Aided Design*, pages 265–269, 1996.

[29] S. Helke, T. Neustupny, and T. Santen. Automating test case generation from z specifications with isabelle. In *LECTURE NOTES IN COMPUTER SCIENCE*, pages 52–71. Springer-Verlag, 1997.

[30] A. Ireland and A. Bundy. Productive use of failure in inductive proof. *Journal of Automated Reasoning*, 16(1-2):79–111, 1996.

[31] A. Ireland and A. Bundy. Automatic Verification of Functions with Accumulating Parameters. *Journal of Functional Programming: Special Issue on Theorem Proving & Functional Programming*, 9(2):225–245, March 1999. A longer version is available from Dept. of Computing and Electrical Engineering, Heriot-Watt University, Research Memo RM/97/11.

[32] M. Johansson. *Automated Discovery of Inductive Lemmas*. PhD thesis, School of Informatics, University of Edinburgh, 2009.

[33] M. Johansson, L. Dixon, and A. Bundy. Conjecture synthesis for inductive theories. *Journal of Automated Reasoning*, 2010. Accepted for publication, to appear.

[34] M. Johansson, L. Dixon, and A. Bundy. Lemma discovery and middle-out reasoning for automated inductive proof. In S. Siegler and N. Wasser, editors, *Induction, Verification and Termination Analysis: Festschrift for Christoph Walther*, volume 6463 of *LNAI*, pages 102–116. Springer-Verlag, 2010.

[35] J.-P. Jouannaud and A. Rubio. The higher-order recursive path ordering. *Logic in Computer Science, Symposium on*, 0:402, 1999.

[36] J.-P. Jouannaud and A. Rubio. Higher-order orderings for normal rewriting. In *In Proc. 17th International Conference on Rewriting Techniques and Applications*, pages 387–399, 2006.

[37] J.-P. Jouannaud and A. Rubio. Polymorphic higher-order recursive path orderings. *J. ACM*, 54:2:1–2:48, March 2007.

[38] D. Kapur and N. A. Sakhanenko. Automatic generation of generalization lemmas for proving properties of tail-recursive definitions. In *In TPHOLs*, pages 136–154, 2003.

[39] A. Krauss. *Automating Recursive Definitions and Termination Proofs in Higher-Order Logic*. PhD thesis, Dept. of Informatics, T. U. München, 2009.

[40] D. B. Lenat. Automated theory formation in mathematics. In *IJCAI'77: Proceedings of the 5th international joint conference on Artificial intelligence*, pages 833–842, San Francisco, CA, USA, 1977. Morgan Kaufmann Publishers Inc.

[41] D. B. Lenat. Automated theory formation in mathematics. In R. Reddy, editor, *Proceedings of IJCAI-77*, pages 833–842. International Joint Conference on Artificial Intelligence, August 1977.

[42] D. B. Lenat. AM: An artificial intelligence approach to discovery in mathematics as heuristic search. In *Knowledge-based systems in Artificial Intelligence*, New York, 1982. McGraw Hill. Also available from Stanford as TechReport AIM 286.

[43] U. Martin and T. Nipkow. Ordered rewriting and confluence. In *Proceedings of the tenth international conference on Automated deduction*, CADE-10, pages 366–380, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

[44] R. Mayr and T. Nipkow. Higher-order rewrite systems and their confluence. *Theoretical Computer Science*, 192:3–29, 1998.

[45] R. McCasland. Mathsaid results: http://dream.inf.ed.ac.uk-/projects/mathsaid/mathsaidresults.html, June 2006.

[46] R. McCasland, A. Bundy, and S. Autexier. Automated discovery of inductive theorems. In R. Matuszewski and A. Zalewska, editors, *From Insight to Proof: Festschrift in Honour of Andrzej Trybulec*, volume 10(23) of *Studies in Logic, Grammar and Rhetoric*, pages 135–149. University of Białystok, 2007.

[47] R. McCasland, A. Bundy, and P. Smith. Ascertaining mathematical theorems. In J. Carette and W. William M. Farmer, editors, *Proceedings of Calculemus 2005*, Newcastle, UK, 2005.

[48] R. McCasland, A. Bundy, and P. Smith. Ascertaining mathematical theorems. *Electronic Notes in Theoretical Computer Science*, 151:21–38, 2006.

[49] W. McCune. The Otter user's guide. Technical Report ANL/90/9, Argonne National Laboratory, 1990.

[50] W. McCune. A Davis-Putnam program and its application to finite first-order model search. Technical Report ANL/MCS-TM-194, Argonne National Laboratoriess, 1994.

[51] W. McCune. Basic test problems: A practical evaluation of some paramodulation strategies, 1996.

[52] W. McCune. MACE 2 reference manual. Technical Report ANL/90/9, Argonne National Laboratories, 2001.

[53] O. Montano-Rivas, R. McCasland, L. Dixon, and A. Bundy. Scheme-based synthesis of inductive theories. In *Proceedings of the 9th Mexican international conference on Advances in artificial intelligence: Part I*, MICAI'10, pages 348–361, Berlin, Heidelberg, 2010. Springer-Verlag.

[54] O. Montano-Rivas, R. McCasland, L. Dixon, and A. Bundy. Scheme-based theorem discovery and concept invention. *Expert Systems with Applications*, 39(2):1637 – 1646, 2012.

[55] T. Nipkow, L. C. Paulson, and M. Wenzel. Isabelle's Logics: HOL, 2000.

[56] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[57] L. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5:363–397, 1989.

[58] L. C. Paulson. A higher-order implementation of rewriting. *Science of Computer Programming*, 3:119–49, 1983.

[59] L. C. Paulson. *Isabelle: A generic theorem prover*. Springer-Verlag, 1994.

[60] G. E. Peterson. Complete sets of reductions with constraints. In *Proceedings of the tenth international conference on Automated deduction*, CADE-10, pages 381–395, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

[61] G. Ritchie. AM: A case study in AI methodology. Technical report, University of Kent, 1981.

[62] J. A. Robinson and A. Voronkov, editors. *Handbook of Automated Reasoning (in 2 volumes)*. Elsevier and MIT Press, 2001.

[63] J. Slaney. FINDER – finder finite domain enumerator version 3.0 – Notes And Guide, 1995.

[64] G. Sutcliffe, Y. Gao, and S. Colton. A Grand Challenge of Theorem Discovery, June 2003.

[65] T. Walsh. A divergence critic for inductive proof. *Journal of Artificial Intelligence Research*, 4:209–235, 1996.

[66] M. Wenzel and T. München. The isabelle/isar reference manual, October 2005.

[67] S. Wilson, J. Fleuriot, and A. Smaill. Automation for dependently typed functional programming. *Fundamenta Informaticae*, 102(2):209–228, 2010.

[68] J. Zhang and H. Zhang. Sem: a system for enumerating models. In *IJCAI*, pages 298–303, 1995.